NUMERICAL METHODS

Jeffrey R. Chasnov
Hong Kong University of Science and Technology

LibreTexts™

Hong Kong University of Science and Technology

Numerical Methods (Chasnov)

Jeffrey R. Chasnov

This text is disseminated via the Open Education Resource (OER) LibreTexts Project (https://LibreTexts.org) and like the thousands of other texts available within this powerful platform, it is freely available for reading, printing, and "consuming."

The LibreTexts mission is to bring together students, faculty, and scholars in a collaborative effort to provide an accessible, and comprehensive platform that empowers our community to develop, curate, adapt, and adopt openly licensed resources and technologies; through these efforts we can reduce the financial burden born from traditional educational resource costs, ensuring education is more accessible for students and communities worldwide.

Most, but not all, pages in the library have licenses that may allow individuals to make changes, save, and print this book. Carefully consult the applicable license(s) before pursuing such effects. Instructors can adopt existing LibreTexts texts or Remix them to quickly build course-specific resources to meet the needs of their students. Unlike traditional textbooks, LibreTexts' web based origins allow powerful integration of advanced features and new technologies to support learning.

LibreTexts is the adaptable, user-friendly non-profit open education resource platform that educators trust for creating, customizing, and sharing accessible, interactive textbooks, adaptive homework, and ancillary materials. We collaborate with individuals and organizations to champion open education initiatives, support institutional publishing programs, drive curriculum development projects, and more.

The LibreTexts libraries are Powered by NICE CXone Expert and was supported by the Department of Education Open Textbook Pilot Project, the California Education Learning Lab, the UC Davis Office of the Provost, the UC Davis Library, the California State University Affordable Learning Solutions Program, and Merlot. This material is based upon work supported by the National Science Foundation under Grant No. 1246120, 1525057, and 1413739.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation nor the US Department of Education.

Have questions or comments? For information about adoptions or adaptions contact info@LibreTexts.org or visit our main website at https://LibreTexts.org.

The LibreTexts name, logo, and book covers are trademarked by LibreTexts, Inc. (a 501(c)(3) not-for-profit organization) and protected against unauthorized use.

This text was compiled on 08/01/2026

- **InfoPage** by Delmar Larsen is licensed Public Domain.

TABLE OF CONTENTS

Licensing

Latex

1: IEEE Arithmetic

- 1.1: Definitions
- 1.2: Numbers with a Decimal or Binary Point
- 1.3: Examples of Binary Numbers
- 1.4: Hex Numbers
- 1.5: 4-bit Unsigned Integers as Hex Numbers
- 1.6: IEEE Single Precision Format
- 1.7: Special Numbers
- 1.8: Examples of Computer Numbers
- 1.9: Inexact Numbers
- 1.10: Machine Epsilon
- 1.11: IEEE Double Precision Format
- 1.12: Roundoff Error Example

2: Root Finding

- 2.1: Bisection Method
- 2.2: Newton's Method
- 2.3: Secant Method
- 2.4: Order of Convergence

3: System of Equations

- 3.1: Gaussian Elimination
- 3.2: LU Decomposition
- 3.3: Partial Pivoting
- 3.4: Operation Counts
- 3.5: System of Nonlinear Equations

4: Least-Squares Approximation

- 4.1: Fitting a Straight Line
- 4.2: Fitting to a Linear Combination of Functions

5: Interpolation

- 5.1: Polynomial Interpolation
- 5.2: Piecewise Linear Interpolation
- 5.3: Cubic Spline Interpolation
- 5.4: Multidimensional Interpolation

6: Integration

- 6.1: Elementary Formulas
- 6.2: Composite Rules
- 6.3: Local Versus Global Error

- 6.4: Adaptive Integration

7: Ordinary Differential Equations

- 7.1: Examples of Analytical Solutions
- 7.2: Numerical Methods - Initial Value Problem
- 7.3: Numerical Methods - Boundary Value Problem
- 7.4: Numerical Methods - Eigenvalue Problem

Index

Detailed Licensing

Licensing

*A detailed breakdown of this resource's licensing can be found in **Back Matter/Detailed Licensing**.*

Latex

Chapter 1

Arithmetic

Definitions

Bit	= 0 or 1	
Byte	= 8 bits	
Word	= Reals:	4 bytes (single precision)
		8 bytes (double precision)
	= Integers:	$1, 2, 4,$ or 8 byte signed
		$1, 2, 4,$ or 8 byte unsigned

Numbers with a decimal or binary point

	□	□	□	□	□	□	□	□	□
Decimal:	10^3	10^2	10^1	10^0		10^{-1}	10^{-2}	10^{-3}	10^{-4}
Binary:	2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}	2^{-4}

Examples of binary numbers

Decimal	Binary
1	1
2	10
3	11
4	100
0.5	0.1
1.5	1.1

numbers

$$\{0, 1, 2, 3, \ldots, 9, 10, 11, 12, 13, 14, 15\} = \{0, 1, 2, 3 \ldots \ldots .. 9, \mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \mathrm{e}, \mathrm{f}\}$$

unsigned integers as hex numbers

Decimal	Binary	Hex
1	0001	1
2	0010	2
3	0011	3
⋮	⋮	⋮
10	1010	a
⋮	⋮	⋮
15	1111	f

single precision format:

where $s =$ sign

$$e = \text{ biased exponent}$$

$$\mathrm{p} = \mathrm{e} - 127 = \text{ exponent}$$

$$1.\mathrm{f} = \text{significand (use binary point)}$$

Special numbers

Smallest exponent: Largest exponent:

reserved reserved

Smallest positive normal number

bin: 00000000000000000000000000000000

hex: 00000000

Examples of computer numbers

What is $1.0, 2.0 \& 1/2$ in hex?

$1.0 = (-1)^0 \times 2^{(127-127)} \times 1.0$

Therefore, $s = 0, e = 01111111, f = 00000000000000000000000$

bin: 00111111100000000000000000000000

hex: 3 f 800000

$2.0 = (-1)^0 \times 2^{(128-127)} \times 1.0$

Therefore, $s = 0, e = 10000000, f = 00000000000000000000000$

bin: 01000000010000000000000000000000

hex: 40000000

$1/2 = (-1)^0 \times 2^{(126-127)} \times 1.0$

Therefore, $s = 0, e = 01111110, f = 00000000000000000000000$

bin: 00111111000000000000000000000000

hex: 3 f00 0000

Inexact numbers

Example:

$$\frac{1}{3} = (-1)^0 \times \frac{1}{4} \times \left(1 + \frac{1}{3}\right),$$

so that $p = e - 127 = -2$ and $e = 125 = 128 - 3$, or in binary, $e = 01111101$. How is $f = 1/3$ represented in binary? To compute binary number, multiply successively by 2 as follows:

$0.333\ldots$	$0.$
$0.666\ldots$	0.0
$1.333\ldots$	0.01
$0.666\ldots$	0.010
$1.333\ldots$	0.0101
etc.	

so that $1/3$ exactly in binary is $0.010101\ldots$ With only 23 bits to represent f, the number is inexact and we have

$$f = 01010101010101010101011$$

where we have rounded to the nearest binary number (here, rounded up). The machine number $1/3$ is then represented as

$$00111110101010101010101010101011$$

or in hex

3eaaaaab.

smallest positive integer that is not exact in single pre- cision

Let N be the smallest positive integer that is not exact. Now, I claim that

$$N-2=2^{23}\times 1.11\ldots 1$$

and

$$N-1=2^{24}\times 1.00\ldots 0$$

The integer N would then require a one-bit in the 2^{-24} position, which is not available. Therefore, the smallest positive integer that is not exact is $2^{24}+1=16777217$. In MATLAB, single (2^{24}) has the same value as single $(2^{24}+1)$. Since single $(2^{24}+1)$ is exactly halfway between the two consecutive machine numbers 2^{24} and $2^{24}+2$, MATLAB rounds to the number with a final zero-bit in f, which is 2^{24}.

Machine epsilon

Machine epsilon (ϵ_{mach}) is the distance between 1 and the next largest number. If $0\le\delta<\epsilon_{\text{mach}}/2$, then $1+\delta=1$ in computer math. Also since

$$x+y=x(1+y/x)$$

if $0\le y/x<\epsilon_{\text{mach}}/2$, then $x+y=x$ in computer math.

Find ϵ_{mach}

The number 1 in the IEEE format is written as

$$1=2^{0}\times 1.000\ldots 0,$$

with $230'$ s following the binary point. The number just larger than 1 has a 1 in the 23 rd position after the decimal point. Therefore,

$$\epsilon_{\text{mach}}=2^{-23}\approx 1.192\times 10^{-7}$$

What is the distance between 1 and the number just smaller than 1? Here, the number just smaller than one can be written as

$$2^{-1}\times 1.111\ldots 1=2^{-1}\left(1+\left(1-2^{-23}\right)\right)=1-2^{-24}$$

Therefore, this distance is $2^{-24}=\epsilon_{\text{mach}}/2$.

The spacing between numbers is uniform between powers of 2, with logarithmic spacing of the powers of 2 . That is, the spacing of numbers between 1 and 2 is 2^{-23}, between 2 and 4 is 2^{-22}, between 4 and 8 is 2^{-21}, etc. This spacing changes for denormal numbers, where the spacing is uniform all the way down to zero.

Find the machine number just greater than 5

A rough estimate would be $5\left(1+\epsilon_{\text{mach}}\right)=5+5\epsilon_{\text{mach}}$, but this is not exact. The exact answer can be found by writing

$$5=2^{2}\left(1+\frac{1}{4}\right)$$

so that the next largest number is

$$2^{2}\left(1+\frac{1}{4}+2^{-23}\right)=5+2^{-21}=5+4\epsilon_{\text{mach}}$$

double precision format

Most computations take place in double precision, where round-off error is reduced, and all of the above calculations in single precision can be repeated for double precision. The format is

where $\mathrm{s}=$ sign

$\mathrm{e}=$ biased exponent

$\mathrm{p} = \mathrm{e} - 1023 =$ exponent

1.f = significand (use binary point)

1.12 Roundoff error example

Consider solving the quadratic equation

$$x^2 + 2bx - 1 = 0$$

where b is a parameter. The quadratic formula yields the two solutions

$$x_{\pm} = -b \pm \sqrt{b^2 + 1}$$

Consider the solution with $b > 0$ and $x > 0$ (the x_+solution) given by

$$x = -b + \sqrt{b^2 + 1}$$

As $b \to \infty$

$$\begin{aligned} x &= -b + \sqrt{b^2 + 1} \\ &= -b + b\sqrt{1 + 1/b^2} \\ &= b\left(\sqrt{1 + 1/b^2} - 1\right) \\ &\approx b\left(1 + \frac{1}{2b^2} - 1\right) \\ &= \frac{1}{2b}. \end{aligned}$$

Now in double precision, realmin $\approx 2.2 \times 10^{-308}$ and we would like x to be accurate to this value before it goes to 0 via denormal numbers. Therefore, x should be computed accurately to $b \approx 1/(2 \times \text{ realmin }) \approx 2 \times 10^{307}$. What happens if we compute (1.1) directly? Then $x = 0$ when $b^2 + 1 = b^2$, or $1 + 1/b^2 = 1$. That is $1/b^2 = \epsilon_{\text{mach}}/2$, or $b = \sqrt{2}/\sqrt{\epsilon_{\text{mach}}} \approx 10^8$. For a subroutine written to compute the solution of a quadratic for a general user, this is not good enough. The way for a software designer to solve this problem is to compute the solution for x as

$$x = \frac{1}{b\left(1 + \sqrt{1 + 1/b^2}\right)}$$

In this form, if $1 + 1/b^2 = 1$, then $x = 1/2b$ which is the correct asymptotic form.

Chapter 2

Finding

Solve $f(x) = 0$ for x, when an explicit analytical solution is impossible.

Bisection Method

The bisection method is the easiest to numerically implement and almost always works. The main disadvantage is that convergence is slow. If the bisection method results in a computer program that runs too slow, then other faster methods may be chosen; otherwise it is a good choice of method.

We want to construct a sequence $x_0, x_1, x_2, \ldots$ that converges to the root $x = r$ that solves $f(x) = 0$. We choose x_0 and x_1 such that $x_0 < r < x_1$. We say that x_0 and x_1 bracket the root. With $f(r) = 0$, we want $f(x_0)$ and $f(x_1)$ to be of opposite sign, so that $f(x_0) f(x_1) < 0$. We then assign x_2 to be the midpoint of x_0 and x_1, that is $x_2 = (x_0 + x_1)/2$, or

$$x_2 = x_0 + \frac{x_1 - x_0}{2}$$

The sign of $f(x_2)$ can then be determined. The value of x_3 is then chosen as either the midpoint of x_0 and x_2 or as the midpoint of x_2 and x_1, depending on whether x_0 and x_2 bracket the root, or x_2 and x_1 bracket the root. The root, therefore, stays bracketed at all

times. The algorithm proceeds in this fashion and is typically stopped when the increment to the left side of the bracket (above, given by $(x_1 - x_0)/2$) is smaller than some required precision.

2.2 Newton's Method

This is the fastest method, but requires analytical computation of the derivative of $f(x)$. Also, the method may not always converge to the desired root.

We can derive Newton's Method graphically, or by a Taylor series. We again want to construct a sequence $x_0, x_1, x_2, ...$ that converges to the root $x = r$. Consider the x_{n+1} member of this sequence, and Taylor series expand $f(x_{n+1})$ about the point x_n. We have

$$f(x_{n+1}) = f(x_n) + (x_{n+1} - x_n) f'(x_n) + \dots .$$

To determine x_{n+1}, we drop the higher-order terms in the Taylor series, and assume $f(x_{n+1}) = 0$. Solving for x_{n+1}, we have

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Starting Newton's Method requires a guess for x_0, hopefully close to the root $x = r$.

Secant Method

The Secant Method is second best to Newton's Method, and is used when a faster convergence than Bisection is desired, but it is too difficult or impossible to take an analytical derivative of the function $f(x)$. We write in place of $f'(x_n)$,

$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$$

Starting the Secant Method requires a guess for both x_0 and x_1.

Estimate $\sqrt{2} = 1.41421356$ using Newton's Method

The $\sqrt{2}$ is the zero of the function $f(x) = x^2 - 2$. To implement Newton's Method, we use $f'(x) = 2x$. Therefore, Newton's Method is the iteration

$$x_{n+1} = x_n - \frac{x_n^2 - 2}{2x_n}$$

We take as our initial guess $x_0 = 1$. Then

$$\begin{aligned} x_1 &= 1 - \frac{-1}{2} = \frac{3}{2} = 1.5 \\ x_2 &= \frac{3}{2} - \frac{\frac{9}{4} - 2}{3} = \frac{17}{12} = 1.416667, \\ x_3 &= \frac{17}{12} - \frac{\frac{17^2}{12^2} - 2}{\frac{17}{6}} = \frac{577}{408} = 1.41426. \end{aligned}$$

Example of fractals using Newton's Method

Consider the complex roots of the equation $f(z) = 0$, where

$$f(z) = z^3 - 1$$

These roots are the three cubic roots of unity. With

$$e^{i2\pi n} = 1, \quad n = 0, 1, 2, ...$$

the three unique cubic roots of unity are given by

$$1, \quad e^{i2\pi/3}, \quad e^{i4\pi/3}$$

With

$$e^{i\theta} = \cos\theta + i \sin\theta,$$

and $\cos(2\pi/3) = -1/2, \sin(2\pi/3) = \sqrt{3}/2$, the three cubic roots of unity are

$$r_1 = 1, \quad r_2 = -\frac{1}{2} + \frac{\sqrt{3}}{2}i, \quad r_3 = -\frac{1}{2} - \frac{\sqrt{3}}{2}i$$

The interesting idea here is to determine which initial values of z_0 in the complex plane converge to which of the three cubic roots of unity.

Newton's method is

$$z_{n+1} = z_n - \frac{z_n^3 - 1}{3z_n^2}$$

If the iteration converges to r_1, we color z_0 red; r_2, blue; r_3, green. The result will be shown in lecture.

Order of convergence

Let r be the root and x_n be the nth approximation to the root. Define the error as

$$\epsilon_n = r - x_n$$

If for large n we have the approximate relationship

$$|\epsilon_{n+1}| = k\,|\epsilon_n|^p\,,$$

with k a positive constant, then we say the root-finding numerical method is of order p. Larger values of p correspond to faster convergence to the root. The order of convergence of bisection is one: the error is reduced by approximately a factor of 2 with each iteration so that

$$|\epsilon_{n+1}| = \frac{1}{2}\,|\epsilon_n|\,.$$

We now find the order of convergence for Newton's Method and for the Secant Method.

Newton's Method

We start with Newton's Method

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Subtracting both sides from r, we have

$$r - x_{n+1} = r - x_n + \frac{f(x_n)}{f'(x_n)}$$

Or

$$\epsilon_{n+1} = \epsilon_n + \frac{f(x_n)}{f'(x_n)}$$

We use Taylor series to expand the functions $f(x_n)$ and $f'(x_n)$ about the root r, using $f(r) = 0$. We have

$$\begin{aligned} f(x_n) &= f(r) + (x_n - r)\,f'(r) + \frac{1}{2}(x_n - r)^2 f''(r) + \dots\,, \\ &= -\epsilon_n f'(r) + \frac{1}{2}\epsilon_n^2 f''(r) + \dots\,; \\ f'(x_n) &= f'(r) + (x_n - r)\,f''(r) + \frac{1}{2}(x_n - r)^2 f'''(r) + \dots\,, \\ &= f'(r) - \epsilon_n f''(r) + \frac{1}{2}\epsilon_n^2 f'''(r) + \dots \end{aligned}$$

To make further progress, we will make use of the following standard Taylor series:

$$\frac{1}{1-\epsilon} = 1 + \epsilon + \epsilon^2 + \dots\,,$$

which converges for $|\epsilon| < 1$. Substituting (2.2) into (2.1), and using (2.3) yields

$$\begin{aligned}
\epsilon_{n+1} &= \epsilon_n + \frac{f(x_n)}{f'(x_n)} \\
&= \epsilon_n + \frac{-\epsilon_n f'(r) + \frac{1}{2}\epsilon_n^2 f''(r) + \ldots}{f'(r) - \epsilon_n f''(r) + \frac{1}{2}\epsilon_n^2 f'''(r) + \ldots} \\
&= \epsilon_n + \frac{-\epsilon_n + \frac{1}{2}\epsilon_n^2 \frac{f''(r)}{f'(r)} + \ldots}{1 - \epsilon_n \frac{f''(r)}{f'(r)} + \ldots} \\
&= \epsilon_n + \left(-\epsilon_n + \frac{1}{2}\epsilon_n^2 \frac{f''(r)}{f'(r)} + \ldots\right)\left(1 + \epsilon_n \frac{f''(r)}{f'(r)} + \ldots\right) \\
&= \epsilon_n + \left(-\epsilon_n + \epsilon_n^2\left(\frac{1}{2}\frac{f''(r)}{f'(r)} - \frac{f''(r)}{f'(r)}\right) + \ldots\right) \\
&= -\frac{1}{2}\frac{f''(r)}{f'(r)}\epsilon_n^2 + \ldots
\end{aligned}$$

Therefore, we have shown that

$$|\epsilon_{n+1}| = k\,|\epsilon_n|^2$$

as $n \to \infty$, with

$$k = \frac{1}{2}\left|\frac{f''(r)}{f'(r)}\right|$$

provided $f'(r) \neq 0$. Newton's method is thus of order 2 at simple roots.

Secant Method

Determining the order of the Secant Method proceeds in a similar fashion. We start with

$$x_{n+1} = x_n - \frac{(x_n - x_{n-1})\,f(x_n)}{f(x_n) - f(x_{n-1})}$$

We subtract both sides from r and make use of

$$\begin{aligned}
x_n - x_{n-1} &= (r - x_{n-1}) - (r - x_n) \\
&= \epsilon_{n-1} - \epsilon_n
\end{aligned}$$

and the Taylor series

$$\begin{aligned}
f(x_n) &= -\epsilon_n f'(r) + \frac{1}{2}\epsilon_n^2 f''(r) + \ldots, \\
f(x_{n-1}) &= -\epsilon_{n-1} f'(r) + \frac{1}{2}\epsilon_{n-1}^2 f''(r) + \ldots,
\end{aligned}$$

so that

$$\begin{aligned}
f(x_n) - f(x_{n-1}) &= (\epsilon_{n-1} - \epsilon_n) f'(r) + \frac{1}{2}\left(\epsilon_n^2 - \epsilon_{n-1}^2\right) f''(r) + \ldots \\
&= (\epsilon_{n-1} - \epsilon_n)\left(f'(r) - \frac{1}{2}(\epsilon_{n-1} + \epsilon_n) f''(r) + \ldots\right)
\end{aligned}$$

We therefore have

$$\begin{aligned}
\epsilon_{n+1} &= \epsilon_n + \frac{-\epsilon_n f'(r) + \frac{1}{2}\epsilon_n^2 f''(r) + \ldots}{f'(r) - \frac{1}{2}(\epsilon_{n-1} + \epsilon_n) f''(r) + \ldots} \\
&= \epsilon_n - \epsilon_n \frac{1 - \frac{1}{2}\epsilon_n \frac{f''(r)}{f'(r)} + \ldots}{1 - \frac{1}{2}(\epsilon_{n-1} + \epsilon_n)\frac{f''(r)}{f'(r)} + \ldots} \\
&= \epsilon_n - \epsilon_n\left(1 - \frac{1}{2}\epsilon_n \frac{f''(r)}{f'(r)} + \ldots\right)\left(1 + \frac{1}{2}(\epsilon_{n-1} + \epsilon_n)\frac{f''(r)}{f'(r)} + \ldots\right) \\
&= -\frac{1}{2}\frac{f''(r)}{f'(r)}\epsilon_{n-1}\epsilon_n + \ldots,
\end{aligned}$$

or to leading order

$$1 \quad |\,f''(r)\,|$$

$$|\epsilon_{n+1}| = \frac{1}{2}\left|\frac{f''(r)}{f'(r)}\right| |\epsilon_{n-1}|\,|\epsilon_n|$$

The order of convergence is not yet obvious from this equation, and to determine the scaling law we look for a solution of the form

$$|\epsilon_{n+1}| = k\,|\epsilon_n|^p\,.$$

From this ansatz, we also have

$$|\epsilon_n| = k\,|\epsilon_{n-1}|^p$$

and therefore

$$|\epsilon_{n+1}| = k^{p+1}\,|\epsilon_{n-1}|^{p^2}$$

Substitution into (2.4) results in

$$k^{p+1}\,|\epsilon_{n-1}|^{p^2} = \frac{k}{2}\left|\frac{f''(r)}{f'(r)}\right| |\epsilon_{n-1}|^{p+1}$$

Equating the coefficient and the power of ϵ_{n-1} results in

$$k^p = \frac{1}{2}\left|\frac{f''(r)}{f'(r)}\right|$$

and

$$p^2 = p + 1$$

The order of convergence of the Secant Method, given by p, therefore is determined to be the positive root of the quadratic equation $p^2 - p - 1 = 0$, or

$$p = \frac{1+\sqrt{5}}{2} \approx 1.618$$

which coincidentally is a famous irrational number that is called The Golden Ratio, and goes by the symbol Φ. We see that the Secant Method has an order of convergence lying between the Bisection Method and Newton's Method.

image

Chapter 3

Systems of equations

Consider the system of n linear equations and n unknowns, given by

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ \vdots & \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned}$$

We can write this system as the matrix equation

$$\mathrm{A}\mathbf{x} = \mathbf{b}$$

with

$$\mathrm{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

Gaussian Elimination

The standard numerical algorithm to solve a system of linear equations is called Gaussian Elimination. We can illustrate this algorithm by example.

Consider the system of equations

$$\begin{aligned} -3x_1 + 2x_2 - x_3 &= -1 \\ 6x_1 - 6x_2 + 7x_3 &= -7 \\ 3x_1 - 4x_2 + 4x_3 &= -6 \end{aligned}$$

To perform Gaussian elimination, we form an Augmented Matrix by combining the matrix A with the column vector b :

$$\begin{pmatrix} -3 & 2 & -1 & -1 \\ 6 & -6 & 7 & -7 \\ 3 & -4 & 4 & -6 \end{pmatrix}$$

Row reduction is then performed on this matrix. Allowed operations are (1) multiply any row by a constant, (2) add multiple of one row to another row, (3) interchange the order of any rows. The goal is to convert the original matrix into an upper-triangular matrix

We start with the first row of the matrix and work our way down as follows. First we multiply the first row by 2 and add it to the second row, and add the first row to the third row:

$$\begin{pmatrix} -3 & 2 & -1 & -1 \\ 0 & -2 & 5 & -9 \\ 0 & -2 & 3 & -7 \end{pmatrix}$$

We then go to the second row. We multiply this row by -1 and add it to the third row:

$$\begin{pmatrix} -3 & 2 & -1 & -1 \\ 0 & -2 & 5 & -9 \\ 0 & 0 & -2 & 2 \end{pmatrix}$$

The resulting equations can be determined from the matrix and are given by

$$\begin{aligned} -3x_1 + 2x_2 - x_3 &= -1 \\ -2x_2 + 5x_3 &= -9 \\ -2x_3 &= 2 \end{aligned}$$

These equations can be solved by backward substitution, starting from the last equation and working backwards. We have

$$\begin{aligned} -2x_3 = 2 &\rightarrow x_3 = -1 \\ -2x_2 = -9 - 5x_3 = -4 &\rightarrow x_2 = 2 \\ -3x_1 = -1 - 2x_2 + x_3 = -6 &\rightarrow x_1 = 2 \end{aligned}$$

Therefore,

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 2 \\ -1 \end{pmatrix}$$

U\) decomposition

The process of Gaussian Elimination also results in the factoring of the matrix A to

$$\mathrm{A} = \mathrm{LU},$$

where L is a lower triangular matrix and U is an upper triangular matrix. Using the same matrix A as in the last section, we show how this factorization is realized. We have

$$\begin{pmatrix} -3 & 2 & -1 \\ 6 & -6 & 7 \\ 3 & -4 & 4 \end{pmatrix} \rightarrow \begin{pmatrix} -3 & 2 & -1 \\ 0 & -2 & 5 \\ 0 & -2 & 3 \end{pmatrix} = \mathrm{M}_1\ \mathrm{A}$$

where

$$\mathrm{M}_1\ \mathrm{A} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} -3 & 2 & -1 \\ 6 & -6 & 7 \\ 3 & -4 & 4 \end{pmatrix} = \begin{pmatrix} -3 & 2 & -1 \\ 0 & -2 & 5 \\ 0 & -2 & 3 \end{pmatrix}$$

Note that the matrix M_1 performs row elimination on the first column. Two times the first row is added to the second row and one times the first row is added to the third row. The entries of the column of M_1 come from $2 = -(6/-3)$ and $1 = -(3/-3)$ as required for row elimination. The number -3 is called the pivot.

The next step is

$$\begin{pmatrix} -3 & 2 & -1 \\ 0 & -2 & 5 \\ 0 & -2 & 3 \end{pmatrix} \rightarrow \begin{pmatrix} -3 & 2 & -1 \\ 0 & -2 & 5 \\ 0 & 0 & -2 \end{pmatrix} = \mathrm{M}_2 \left(\mathrm{M}_1 \ \mathrm{A}\right)$$

where

$$\mathrm{M}_2 \left(\mathrm{M}_1 \ \mathrm{A}\right) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} -3 & 2 & -1 \\ 0 & -2 & 5 \\ 0 & -2 & 3 \end{pmatrix} = \begin{pmatrix} -3 & 2 & -1 \\ 0 & -2 & 5 \\ 0 & 0 & -2 \end{pmatrix}$$

Here, M_2 multiplies the second row by $-1 = -(-2/-2)$ and adds it to the third row. The pivot is -2.

We now have

$$\mathrm{M}_2 \mathrm{M}_1 \ \mathrm{A} = \mathrm{U}$$

or

$$\mathrm{A} = \mathrm{M}_1^{-1} \mathrm{M}_2^{-1} \mathrm{U}$$

The inverse matrices are easy to find. The matrix M_1 multiples the first row by 2 and adds it to the second row, and multiplies the first row by 1 and adds it to the third row. To invert these operations, we need to multiply the first row by -2 and add it to the second row, and multiply the first row by -1 and add it to the third row. To check, with

$$\mathrm{M}_1 \mathrm{M}_1^{-1} = \mathrm{I}$$

we have

$$\begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Similarly,

$$\mathrm{M}_2^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

Therefore,

$$\mathrm{L} = \mathrm{M}_1^{-1} \mathrm{M}_2^{-1}$$

is given by

$$L = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -1 & 1 & 1 \end{pmatrix}$$

which is lower triangular. The off-diagonal elements of M_1^{-1} and M_2^{-1} are simply combined to form L. Our LU decomposition is therefore

$$\begin{pmatrix} -3 & 2 & -1 \\ 6 & -6 & 7 \\ 3 & -4 & 4 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -1 & 1 & 1 \end{pmatrix} \begin{pmatrix} -3 & 2 & -1 \\ 0 & -2 & 5 \\ 0 & 0 & -2 \end{pmatrix}.$$

Another nice feature of the LU decomposition is that it can be done by overwriting A, therefore saving memory if the matrix A is very large.

The LU decomposition is useful when one needs to solve $A\mathbf{x} = \mathbf{b}$ for $\mathbf{x}$ when A is fixed and there are many different b's. First one determines L and U using Gaussian elimination. Then one writes

$$(\mathrm{LU})\mathbf{x} = \mathrm{L}(\mathrm{U}\mathbf{x}) = \mathbf{b}.$$

We let

$$\mathbf{y} = \mathbf{U}\mathbf{x}$$

and first solve

$$\mathrm{L}\mathbf{y} = \mathbf{b}$$

for y by forward substitution. We then solve

$$\mathrm{U}\mathbf{x} = \mathbf{y}$$

for $\mathbf{x}$ by backward substitution. When we count operations, we will see that solving $(\mathrm{LU})\mathbf{x} = \mathbf{b}$ is significantly faster once L and U are in hand than solving $\mathrm{A}\mathbf{x} = \mathbf{b}$ directly by Gaussian elimination.

We now illustrate the solution of $LU\mathbf{x} = \mathbf{b}$ using our previous example, where

$$\mathrm{L} = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -1 & 1 & 1 \end{pmatrix}, \quad \mathrm{U} = \begin{pmatrix} -3 & 2 & -1 \\ 0 & -2 & 5 \\ 0 & 0 & -2 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} -1 \\ -7 \\ -6 \end{pmatrix}$$

With $\mathbf{y} = \mathbf{U}\mathbf{x}$, we first solve $\mathbf{L}\mathbf{y} = \mathbf{b}$, that is

$$\begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -1 & 1 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} -1 \\ -7 \\ -6 \end{pmatrix}$$

Using forward substitution

$$\begin{aligned} y_1 &= -1 \\ y_2 &= -7 + 2y_1 = -9 \\ y_3 &= -6 + y_1 - y_2 = 2 \end{aligned}$$

We now solve $Ux = y$, that is

$$\begin{pmatrix} -3 & 2 & -1 \\ 0 & -2 & 5 \\ 0 & 0 & -2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} -1 \\ -9 \\ 2 \end{pmatrix}$$

Using backward substitution,

$$\begin{aligned} -2x_3 &= 2 \rightarrow x_3 = -1 \\ -2x_2 &= -9 - 5x_3 = -4 \rightarrow x_2 = 2 \\ -3x_1 &= -1 - 2x_2 + x_3 = -6 \rightarrow x_1 = 2 \end{aligned}$$

and we have once again determined

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 2 \\ -1 \end{pmatrix}$$

Partial pivoting

When performing Gaussian elimination, the diagonal element that one uses during the elimination procedure is called the pivot. To obtain the correct multiple, one uses the pivot as the divisor to the elements below the pivot. Gaussian elimination in this form will fail if the pivot is zero. In this situation, a row interchange must be performed.

Even if the pivot is not identically zero, a small value can result in big roundoff errors. For very large matrices, one can easily lose all accuracy in the solution. To avoid these round-off errors arising from small pivots, row interchanges are made, and this

technique is called partial pivoting (partial pivoting is in contrast to complete pivoting, where both rows and columns are interchanged). We will illustrate by example the LU decomposition using partial pivoting. Consider

$$A = \begin{pmatrix} -2 & 2 & -1 \\ 6 & -6 & 7 \\ 3 & -8 & 4 \end{pmatrix}$$

We interchange rows to place the largest element (in absolute value) in the pivot, or a_{11}, position. That is,

$$\mathrm{A} \rightarrow \begin{pmatrix} 6 & -6 & 7 \\ -2 & 2 & -1 \\ 3 & -8 & 4 \end{pmatrix} = \mathrm{P}_{12}\ \mathrm{A}$$

where

$$P_{12} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

is a permutation matrix that when multiplied on the left interchanges the first and second rows of a matrix. Note that $P_{12}^{-1} = P_{12}$. The elimination step is then

$$\mathrm{P}_{12}\ \mathrm{A} \rightarrow \begin{pmatrix} 6 & -6 & 7 \\ 0 & 0 & 4/3 \\ 0 & -5 & 1/2 \end{pmatrix} = \mathrm{M}_1\mathrm{P}_{12}\ \mathrm{A}$$

where

$$M_1 = \begin{pmatrix} 1 & 0 & 0 \\ 1/3 & 1 & 0 \\ -1/2 & 0 & 1 \end{pmatrix}$$

The final step requires one more row interchange:

$$\mathrm{M}_1\mathrm{P}_{12}\ \mathrm{A} \rightarrow \begin{pmatrix} 6 & -6 & 7 \\ 0 & -5 & 1/2 \\ 0 & 0 & 4/3 \end{pmatrix} = \mathrm{P}_{23}\mathrm{M}_1\mathrm{P}_{12}\ \mathrm{A} = \mathrm{U}$$

Since the permutation matrices given by P are their own inverses, we can write our result as

$$\left(P_{23}M_1P_{23}\right) P_{23}P_{12}A = U$$

Multiplication of M on the left by P interchanges rows while multiplication on the right by P interchanges columns. That is,

$$P_{23}\begin{pmatrix} 1 & 0 & 0 \\ 1/3 & 1 & 0 \\ -1/2 & 0 & 1 \end{pmatrix} P_{23} = \begin{pmatrix} 1 & 0 & 0 \\ -1/2 & 0 & 1 \\ 1/3 & 1 & 0 \end{pmatrix} P_{23} = \begin{pmatrix} 1 & 0 & 0 \\ -1/2 & 1 & 0 \\ 1/3 & 0 & 1 \end{pmatrix}$$

The net result on M_1 is an interchange of the nondiagonal elements $1/3$ and $-1/2$.

We can then multiply by the inverse of $(\mathrm{P}_{23}\mathrm{M}_1\mathrm{P}_{23})$ to obtain

$$P_{23}P_{12}A = \left(P_{23}M_1P_{23}\right)^{-1} U$$

which we write as

$$\mathrm{PA} = \mathrm{LU}$$

Instead of L, MATLAB will write this as

$$\mathrm{A} = \left(\mathrm{P}^{-1}\ \mathrm{L}\right) \mathrm{U}$$

For convenience, we will just denote $\left(\mathrm{P}^{-1}\ \mathrm{L}\right)$ by L, but by L here we mean a permutated lower triangular matrix.

For example, in MATLAB, to solve $A\mathrm{x} = \mathrm{b}$ for x using Gaussian elimination, one types

$$\mathrm{x} = \mathrm{A}\backslash\mathrm{b}$$

where \solves for x using the most efficient algorithm available, depending on the form of A. If A is a general $n \times n$ matrix, then first the LU decomposition of A is found using partial pivoting, and then x is determined from permuted forward and backward substitution. If A is upper or lower triangular, then forward or backward substitution (or their permuted version) is used directly.

If there are many different right-hand-sides, one would first directly find the LU decomposition of A using a function call, and then solve using . That is, one would iterate for different b′ s the following expressions:

$$\begin{aligned}[\mathrm{LU}] &= lu(\ \mathrm{A}) \\ \mathrm{y} &= \mathrm{L}\backslash\mathrm{b} \\ \mathrm{x} &= \mathrm{U}\backslash y\end{aligned}$$

where the second and third lines can be shortened to

$$\mathrm{x} = \mathrm{U}\backslash(\mathrm{L}\backslash\mathrm{b})$$

where the parenthesis are required. In lecture, I will demonstrate these solutions in MATLAB using the matrix $A = [-2, 2, -1; 6, -6, 7; 3, -8, 4]$; which is the example in the notes.

3.4 Operation counts

To estimate how much computational time is required for an algorithm, one can count the number of operations required (multiplications, divisions, additions and subtractions). Usually, what is of interest is how the algorithm scales with the size of the problem. For example, suppose one wants to multiply two full $n \times n$ matrices. The calculation of each element requires n multiplications and $n-1$ additions, or say $2n-1$ operations. There are n^2 elements to compute so that the total operation count is $n^2(2n-1)$. If n is large, we might want to know what will happen to the computational time if n is doubled. What matters most is the fastest-growing, leading-order term in the operation count. In this matrix multiplication example, the operation count is $n^2(2n-1) = 2n^3 - n^2$, and the leading-order term is $2n^3$. The factor of 2 is unimportant for the scaling, and we say that the algorithm scales like $\mathrm{O}\left(n^3\right)$, which is read as "big Oh of n cubed." When using the big-Oh notation, we will drop both lower-order terms and constant multipliers.

The big-Oh notation tells us how the computational time of an algorithm scales. For example, suppose that the multiplication of two large $n \times n$ matrices took a computational time of T_n seconds. With the known operation count going like $\mathrm{O}\left(n^3\right)$, we can write

$$T_n = kn^3$$

for some unknown constant k. To determine how much longer the multiplication of two $2n \times 2n$ matrices will take, we write

$$\begin{aligned}T_{2n} &= k(2n)^3 \\ &= 8kn^3 \\ &= 8T_n\end{aligned}$$

so that doubling the size of the matrix is expected to increase the computational time by a factor of $2^3 = 8$.

Running MATLAB on my computer, the multiplication of two 2048×2048 matrices took about $0.75\mathrm{sec}$. The multiplication of two 4096×4096 matrices took about $6\mathrm{sec}$, which is 8 times longer. Timing of code in MATLAB can be found using the built-in stopwatch functions tic and toc.

What is the operation count and therefore the scaling of Gaussian elimination? Consider an elimination step with the pivot in the i th row and i th column. There are both $n-i$ rows below the pivot and $n-i$ columns to the right of the pivot. To perform elimination of one row, each matrix element to the right of the pivot must be multiplied by a factor and added to the row underneath. This must be done for all the rows. There are therefore $(n-i)(n-i)$ multiplication-additions to be done for this pivot. Since we are interested in only the scaling of the algorithm, I will just count a multiplication-addition as one operation.

To find the total operation count, we need to perform elimination using $n-1$ pivots, so that

$$\begin{aligned}\text{op. counts} &= \sum_{i=1}^{n-1}(n-i)^2 \\ &= (n-1)^2 + (n-2)^2 + \ldots (1)^2 \\ &= \sum_{i=1}^{n-1} i^2\end{aligned}$$

Three summation formulas will come in handy. They are

$$\sum_{i=1}^{n} 1 = n$$
$$\sum_{i=1}^{n} i = \frac{1}{2} n(n+1)$$
$$\sum_{i=1}^{n} i^2 = \frac{1}{6} n(2n+1)(n+1)$$

which can be proved by mathematical induction, or derived by some tricks.

The operation count for Gaussian elimination is therefore

$$\begin{aligned} \text{op. counts} &= \sum_{i=1}^{n-1} i^2 \\ &= \frac{1}{6}(n-1)(2n-1)(n) \end{aligned}$$

The leading-order term is therefore $n^3/3$, and we say that Gaussian elimination scales like $\mathrm{O}\left(n^3\right)$. Since LU decomposition with partial pivoting is essentially Gaussian elimination, that algorithm also scales like $\mathrm{O}\left(n^3\right)$. However, once the LU decomposition of a matrix A is known, the solution of $\mathrm{A}\mathbf{x} = \mathbf{b}$ can proceed by a forward and backward substitution. How does a backward substitution, say, scale? For backward substitution, the matrix equation to be solved is of the form

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n-1} & a_{1,n} \\ 0 & a_{2,2} & \cdots & a_{2,n-1} & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & a_{n-1,n-1} & a_{n-1,n} \\ 0 & 0 & \cdots & 0 & a_{n,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n-1} \\ b_n \end{pmatrix}$$

The solution for x_i is found after solving for x_j with $j > i$. The explicit solution for x_i is given by

$$x_i = \frac{1}{a_{i,i}} \left(b_i - \sum_{j=i+1}^{n} a_{i,j} x_j \right)$$

The solution for x_i requires $n - i + 1$ multiplication-additions, and since this must be done for n such i' s, we have

$$\begin{aligned} \text{op. counts} &= \sum_{i=1}^{n} n - i + 1 \\ &= n + (n-1) + \cdots + 1 \\ &= \sum_{i=1}^{n} i \\ &= \frac{1}{2} n(n+1) \end{aligned}$$

The leading-order term is $n^2/2$ and the scaling of backward substitution is $\mathrm{O}\left(n^2\right)$. After the LU decomposition of a matrix A is found, only a single forward and backward substitution is required to solve $A\mathbf{x} = \mathbf{b}$, and the scaling of the algorithm to solve this matrix equation is therefore still $\mathrm{O}\left(n^2\right)$. For large n, one should expect that solving $\mathrm{A}\mathbf{x} = \mathbf{b}$ by a forward and backward substitution should be substantially faster than a direct solution using Gaussian elimination.

3.5 System of nonlinear equations

A system of nonlinear equations can be solved using a version of Newton's Method. We illustrate this method for a system of two equations and two unknowns. Suppose that we want to solve

$$f(x, y) = 0, \quad g(x, y) = 0$$

for the unknowns x and y. We want to construct two simultaneous sequences $x_0, x_1, x_2, \ldots$ and $y_0, y_1, y_2, \ldots$ that converge to the roots. To construct these sequences, we Taylor series expand $f\left(x_{n+1}, y_{n+1}\right)$ and $g\left(x_{n+1}, y_{n+1}\right)$ about the point $\left(x_n, y_n\right)$. Using the partial

derivatives $f_x = \partial f/\partial x, f_y = \partial f/\partial y$, etc., the twodimensional Taylor series expansions, displaying only the linear terms, are given by

$$f(x_{n+1}, y_{n+1}) = f(x_n, y_n) + (x_{n+1} - x_n) f_x(x_n, y_n) + (y_{n+1} - y_n) f_y(x_n, y_n) + \ldots$$

$$g(x_{n+1}, y_{n+1}) = g(x_n, y_n) + (x_{n+1} - x_n) g_x(x_n, y_n) + (y_{n+1} - y_n) g_y(x_n, y_n) + \ldots$$

To obtain Newton's method, we take $f(x_{n+1}, y_{n+1}) = 0, g(x_{n+1}, y_{n+1}) = 0$ and drop higher-order terms above linear. Although one can then find a system of linear equations for x_{n+1} and y_{n+1}, it is more convenient to define the variables

$$\Delta x_n = x_{n+1} - x_n, \quad \Delta y_n = y_{n+1} - y_n.$$

The iteration equations will then be given by

$$x_{n+1} = x_n + \Delta x_n, \quad y_{n+1} = y_n + \Delta y_n$$

and the linear equations to be solved for Δx_n and Δy_n are given by

$$\begin{pmatrix} f_x & f_y \\ g_x & g_y \end{pmatrix} \begin{pmatrix} \Delta x_n \\ \Delta y_n \end{pmatrix} = \begin{pmatrix} -f \\ -g \end{pmatrix}$$

where f, g, f_x, f_y, g_x, and g_y are all evaluated at the point (x_n, y_n). The twodimensional case is easily generalized to n dimensions. The matrix of partial derivatives is called the Jacobian Matrix.

We illustrate Newton's Method by finding the steady state solution of the Lorenz equations, given by

$$\begin{aligned} \sigma(y - x) &= 0 \\ rx - y - xz &= 0 \\ xy - bz &= 0 \end{aligned}$$

where x, y, and z are the unknown variables and σ, r, and b are the known parameters. Here, we have a three-dimensional homogeneous system $f = 0, g = 0$, and $h = 0$, with

$$\begin{aligned} f(x, y, z) &= \sigma(y - x) \\ g(x, y, z) &= rx - y - xz \\ h(x, y, z) &= xy - bz. \end{aligned}$$

The partial derivatives can be computed to be

$$\begin{aligned} f_x &= -\sigma, & f_y &= \sigma, & f_z &= 0, \\ g_x &= r - z, & g_y &= -1, & g_z &= -x, \\ h_x &= y, & h_y &= x, & h_z &= -b. \end{aligned}$$

The iteration equation is therefore

$$\begin{pmatrix} -\sigma & \sigma & 0 \\ r - z_n & -1 & -x_n \\ y_n & x_n & -b \end{pmatrix} \begin{pmatrix} \Delta x_n \\ \Delta y_n \\ \Delta z_n \end{pmatrix} = - \begin{pmatrix} \sigma(y_n - x_n) \\ rx_n - y_n - x_n z_n \\ x_n y_n - b z_n \end{pmatrix}$$

with

$$\begin{aligned} x_{n+1} &= x_n + \Delta x_n \\ y_{n+1} &= y_n + \Delta y_n \\ z_{n+1} &= z_n + \Delta z_n \end{aligned}$$

The MATLAB program that solves this system is contained in newton_system.m.

image

Chapter 4

Least-squares approximation

The method of least-squares is commonly used to fit a parameterized curve to experimental data. In general, the fitting curve is not expected to pass through the data points, making this problem substantially different from the one of interpolation.

We consider here only the simplest case of the same experimental error for all the data points. Let the data to be fitted be given by (x_i, y_i), with $i = 1$ to n.

Fitting a straight line

Suppose the fitting curve is a line. We write for the fitting curve

$$y(x) = \alpha x + \beta.$$

The distance r_i from the data point (x_i, y_i) and the fitting curve is given by

$$\begin{aligned} r_i &= y_i - y\left(x_i\right) \\ &= y_i - \left(\alpha x_i + \beta\right). \end{aligned}$$

A least-squares fit minimizes the sum of the squares of the r_i 's. This minimum can be shown to result in the most probable values of α and β.

We define

$$\begin{aligned} \rho &= \sum_{i=1}^{n} r_i^2 \\ &= \sum_{i=1}^{n} \left(y_i - \left(\alpha x_i + \beta\right)\right)^2 \end{aligned}$$

To minimize ρ with respect to α and β, we solve

$$\frac{\partial \rho}{\partial \alpha} = 0, \quad \frac{\partial \rho}{\partial \beta} = 0$$

Taking the partial derivatives, we have

$$\begin{aligned} \frac{\partial \rho}{\partial \alpha} &= \sum_{i=1}^{n} 2\left(-x_i\right)\left(y_i - \left(\alpha x_i + \beta\right)\right) = 0 \\ \frac{\partial \rho}{\partial \beta} &= \sum_{i=1}^{n} 2(-1)\left(y_i - \left(\alpha x_i + \beta\right)\right) = 0 \end{aligned}$$

These equations form a system of two linear equations in the two unknowns α and β, which is evident when rewritten in the form

$$\begin{aligned} \alpha \sum_{i=1}^{n} x_i^2 + \beta \sum_{i=1}^{n} x_i &= \sum_{i=1}^{n} x_i y_i \\ \alpha \sum_{i=1}^{n} x_i + \beta n &= \sum_{i=1}^{n} y_i \end{aligned}$$

These equations can be solved either analytically, or numerically in MATLAB, where the matrix form is

$$\begin{pmatrix} \sum_{i=1}^{n} & x_i^2 & \sum_{i=1}^{n} \\ \sum_{i=1}^{n} & x_i & n \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{n} x_i y_i \\ \sum_{i=1}^{n} y_i \end{pmatrix}.$$

A proper statistical treatment of this problem should also consider an estimate of the errors in α and β as well as an estimate of the goodness-of-fit of the data to the model. We leave these further considerations to a statistics class.

Fitting to a linear combination of functions

Consider the general fitting function

$$y(x) = \sum_{j=1}^{m} c_j f_j(x)$$

where we assume m functions $f_j(x)$. For example, if we want to fit a cubic polynomial to the data, then we would have $m = 4$ and take $f_1 = 1, f_2 = x, f_3 = x^2$ and $f_4 = x^3$. Typically, the number of functions f_j is less than the number of data points; that is, $m < n$, so that a direct attempt to solve for the c_j' s such that the fitting function exactly passes through the n data points would result in n equations and m unknowns. This would be an over-determined linear system that in general has no solution.

We now define the vectors

$$\mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}, \quad \mathbf{c} = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{pmatrix}$$

and the n-by-m matrix

$$\mathrm{A} = \begin{pmatrix} f_1(x_1) & f_2(x_1) & \cdots & f_m(x_1) \\ f_1(x_2) & f_2(x_2) & \cdots & f_m(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_n) & f_2(x_n) & \cdots & f_m(x_n) \end{pmatrix}$$

With $m < n$, the equation $A\mathbf{c} = \mathbf{y}$ is over-determined. We let

$$\mathbf{r} = \mathbf{y} - \mathrm{A}\mathbf{c}$$

be the residual vector, and let

$$\rho = \sum_{i=1}^{n} r_i^2$$

The method of least squares minimizes ρ with respect to the components of c. Now, using T to signify the transpose of a matrix, we have

$$\begin{aligned} \rho &= \mathbf{r}^T\mathbf{r} \\ &= (\mathbf{y} - \mathrm{A}\mathbf{c})^T(\mathbf{y} - \mathrm{A}\mathbf{c}) \\ &= \mathbf{y}^T\mathbf{y} - \mathbf{c}^T\ \mathrm{A}^T\mathbf{y} - \mathbf{y}^T\ \mathrm{A}\mathbf{c} + \mathbf{c}^T\ \mathrm{A}^T\ \mathrm{A}\mathbf{c}. \end{aligned}$$

Since ρ is a scalar, each term in the above expression must be a scalar, and since the transpose of a scalar is equal to the scalar, we have

$$\mathbf{c}^T\ \mathrm{A}^T\mathbf{y} = \left(\mathbf{c}^T\ \mathrm{A}^T\mathbf{y}\right)^T = \mathbf{y}^T\ \mathrm{A}\mathbf{c}.$$

Therefore,

$$\rho = \mathbf{y}^T\mathbf{y} - 2\mathbf{y}^T\ \mathrm{A}\mathbf{c} + \mathbf{c}^T\ \mathrm{A}^T\ \mathrm{A}\mathbf{c}.$$

To find the minimum of ρ, we will need to solve $\partial\rho/\partial c_j = 0$ for $j = 1, \ldots, m$. To take the derivative of ρ, we switch to a tensor notation, using the Einstein summation convention, where repeated indices are summed over their allowable range. We can write

$$\rho = y_i y_i - 2y_i\ \mathrm{A}_{ik}c_k + c_i\ \mathrm{A}_{ik}^T\ \mathrm{A}_{kl}c_l$$

Taking the partial derivative, we have

$$\frac{\partial\rho}{\partial c_j} = -2y_i\ \mathrm{A}_{ik}\frac{\partial c_k}{\partial c_j} + \frac{\partial c_i}{\partial c_j}\ \mathrm{A}_{ik}^T\ \mathrm{A}_{kl}c_l + c_i\ \mathrm{A}_{ik}^T\ \mathrm{A}_{kl}\frac{\partial c_l}{\partial c_j}$$

Now,

$$\frac{\partial c_i}{\partial c_j} = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{otherwise} \end{cases}$$

Therefore,

$$\frac{\partial\rho}{\partial c_j} = -2y_i\ \mathrm{A}_{ij} + \mathrm{A}_{jk}^T\ \mathrm{A}_{kl}c_l + c_i\ \mathrm{A}_{ik}^T\ \mathrm{A}_{kj}$$

Now,

$$\begin{aligned} c_i \ \mathrm{A}_{ik}^T \ \mathrm{A}_{kj} &= c_i \ \mathrm{A}_{ki} \ \mathrm{A}_{kj} \\ &= \mathrm{A}_{kj} \ \mathrm{A}_{ki} c_i \\ &= \mathrm{A}_{jk}^T \ \mathrm{A}_{ki} c_i \\ &= \mathrm{A}_{jk}^T \ \mathrm{A}_{kl} c_l \end{aligned}$$

Therefore,

$$\frac{\partial \rho}{\partial c_j} = -2 y_i \ \mathrm{A}_{ij} + 2 \ \mathrm{A}_{jk}^T \ \mathrm{A}_{kl} c_l$$

With the partials set equal to zero, we have

$$\mathrm{A}_{jk}^T \ \mathrm{A}_{kl} c_l = y_i \ \mathrm{A}_{ij}$$

or

$$\mathrm{A}_{jk}^T \ \mathrm{A}_{kl} c_l = \mathrm{A}_{ji}^T y_i$$

In vector notation, we have

$$\mathrm{A}^T \ \mathrm{A}\mathbf{c} = \mathrm{A}^T \mathbf{y}.$$

Equation (4.2) is the so-called normal equation, and can be solved for c by Gaussian elimination using the MATLAB backslash operator. After constructing the matrix A given by (4.1), and the vector y from the data, one can code in MATLAB

$$c = (A'A) \setminus (A'y)$$

But in fact the MATLAB back slash operator will automatically solve the normal equations when the matrix A is not square, so that the MATLAB code

$$c = A \setminus y$$

yields the same result.

image

Chapter 5

Interpolation

Consider the following problem: Given the values of a known function $y = f(x)$ at a sequence of ordered points $x_0, x_1, \ldots, x_n$, find $f(x)$ for arbitrary x. When $x_0 \leq x \leq x_n$, the problem is called interpolation. When $x < x_0$ or $x > x_n$ the problem is called extrapolation.

With $y_i = f(x_i)$, the problem of interpolation is basically one of drawing a smooth curve through the known points $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$. This is not the same problem as drawing a smooth curve that approximates a set of data points that have experimental error. This latter problem is called least-squares approximation.

Here, we will consider three interpolation algorithms: (1) polynomial interpolation; (2) piecewise linear interpolation, and; (3) cubic spline interpolation.

Polynomial interpolation

The $n+1$ points $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$ can be interpolated by a unique polynomial of degree n. When $n = 1$, the polynomial is a linear function; when $n = 2$, the polynomial is a quadratic function. There are three standard algorithms that can be used to construct this unique interpolating polynomial, and we will present all three here, not so much because they are all useful, but because it is interesting to learn how these three algorithms are constructed.

When discussing each algorithm, we define $P_n(x)$ to be the unique nth degree polynomial that passes through the given $n+1$ data points.

Vandermonde polynomial

This Vandermonde polynomial is the most straightforward construction of the interpolating polynomial $P_n(x)$. We simply write

$$P_n(x) = c_0 x^n + c_1 x^{n-1} + \cdots + c_n.$$

Then we can immediately form $n+1$ linear equations for the $n+1$ unknown coefficients $c_0, c_1, \ldots, c_n$ using the $n+1$ known points:

$$\begin{aligned} y_0 &= c_0 x_0^n + c_1 x_0^{n-1} + \cdots + c_{n-1} x_0 + c_n \\ y_2 &= c_0 x_1^n + c_1 x_1^{n-1} + \cdots + c_{n-1} x_1 + c_n \\ &\vdots \\ y_n &= c_0 x_n^n + c_1 x_n^{n-1} + \cdots + c_{n-1} x_n + c_n \end{aligned}$$

The system of equations in matrix form is

$$\begin{pmatrix} x_0^n & x_0^{n-1} & \cdots & x_0 & 1 \\ x_1^n & x_1^{n-1} & \cdots & x_1 & 1 \\ \vdots & \vdots & \ddots & \vdots & \\ x_n^n & x_n^{n-1} & \cdots & x_n & 1 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix}$$

The matrix is called the Vandermonde matrix, and can be constructed using the MATLAB function vander.m. The system of linear equations can be solved in MATLAB using the \operator, and the MATLAB function polyval.m can used to interpolate using the c coefficients. I will illustrate this in class and place the code on the website.

Lagrange polynomial

The Lagrange polynomial is the most clever construction of the interpolating polynomial $P_n(x)$, and leads directly to an analytical formula. The Lagrange polynomial is the sum of $n+1$ terms and each term is itself a polynomial of degree n. The full polynomial is therefore of degree n. Counting from 0 , the i th term of the Lagrange polynomial is constructed by requiring it to be zero at x_j with $j \neq i$, and equal to y_i when $j = i$. The polynomial can be written as

$$\begin{aligned} P_n(x) = \tfrac{(x-x_1)(x-x_2)\cdots(x-x_n)y_0}{(x_0-x_1)(x_0-x_2)\cdots(x_0-x_n)} + \tfrac{(x-x_0)(x-x_2)\cdots(x-x_n)y_1}{(x_1-x_0)(x_1-x_2)\cdots(x_1-x_n)} \\ + \cdots + \tfrac{(x-x_0)(x-x_1)\cdots(x-x_{n-1})y_n}{(x_n-x_0)(x_n-x_1)\cdots(x_n-x_{n-1})}. \end{aligned}$$

It can be clearly seen that the first term is equal to zero when $x = x_1, x_2, \ldots, x_n$ and equal to y_0 when $x = x_0$; the second term is equal to zero when $x = x_0, x_2, \ldots x_n$ and equal to y_1 when $x = x_1$; and the last term is equal to zero when $x = x_0, x_1, \ldots x_{n-1}$ and equal to y_n when $x = x_n$. The uniqueness of the interpolating polynomial implies that the Lagrange polynomial must be the interpolating polynomial.

Newton polynomial

The Newton polynomial is somewhat more clever than the Vandermonde polynomial because it results in a system of linear equations that is lower triangular, and therefore can be solved by forward substitution. The interpolating polynomial is written in the form

$$P_n(x) = c_0 + c_1 (x - x_0) + c_2 (x - x_0)(x - x_1) + \cdots + c_n (x - x_0) \cdots (x - x_{n-1})$$

which is clearly a polynomial of degree n. The $n+1$ unknown coefficients given by the c' s can be found by substituting the points (x_i, y_i) for $i = 0, \ldots, n$:

$$\begin{aligned} y_0 &= c_0 \\ y_1 &= c_0 + c_1 (x_1 - x_0) \\ y_2 &= c_0 + c_1 (x_2 - x_0) + c_2 (x_2 - x_0)(x_2 - x_1) \\ &\vdots \\ y_n &= c_0 + c_1 (x_n - x_0) + c_2 (x_n - x_0)(x_n - x_1) + \cdots + c_n (x_n - x_0) \cdots (x_n - x_{n-1}) \end{aligned}$$

This system of linear equations is lower triangular as can be seen from the matrix form

$$\begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 1 & (x_1 - x_0) & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & (x_n - x_0) & (x_n - x_0)(x_n - x_1) & \cdots & (x_n - x_0) \cdots (x_n - x_{n-1}) \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{pmatrix}$$

and so theoretically can be solved faster than the Vandermonde polynomial. In practice, however, there is little difference because polynomial interpolation is only useful when the number of points to be interpolated is small.

Piecewise linear interpolation

Instead of constructing a single global polynomial that goes through all the points, one can construct local polynomials that are then connected together. In the the section following this one, we will discuss how this may be done using cubic polynomials. Here, we discuss the simpler case of linear polynomials. This is the default interpolation typically used when plotting data.

Suppose the interpolating function is $y = g(x)$, and as previously, there are $n + 1$ points to interpolate. We construct the function $g(x)$ out of n local linear polynomials. We write

$$g(x) = g_i(x), \quad \text{for } x_i \leq x \leq x_{i+1}$$

where

$$g_i(x) = a_i (x - x_i) + b_i$$

and $i = 0, 1, \ldots, n-1$.

We now require $y = g_i(x)$ to pass through the endpoints (x_i, y_i) and (x_{i+1}, y_{i+1}). We have

$$\begin{aligned} y_i &= b_i \\ y_{i+1} &= a_i (x_{i+1} - x_i) + b_i. \end{aligned}$$

Therefore, the coefficients of $g_i(x)$ are determined to be

$$a_i = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}, \quad b_i = y_i$$

Although piecewise linear interpolation is widely used, particularly in plotting routines, it suffers from a discontinuity in the derivative at each point. This results in a function which may not look smooth if the points are too widely spaced. We next consider a more challenging algorithm that uses cubic polynomials.

Cubic spline interpolation

The $n + 1$ points to be interpolated are again

$$(x_0, y_0), (x_1, y_1), \ldots (x_n, y_n)$$

Here, we use n piecewise cubic polynomials for interpolation,

$$g_i(x) = a_i (x - x_i)^3 + b_i (x - x_i)^2 + c_i (x - x_i) + d_i, \quad i = 0, 1, \ldots, n-1,$$

with the global interpolation function written as

$$g(x) = g_i(x), \quad \text{for } x_i \leq x \leq x_{i+1}.$$

To achieve a smooth interpolation we impose that $g(x)$ and its first and second derivatives are continuous. The requirement that $g(x)$ is continuous (and goes through all $n + 1$ points) results in the two constraints

$$\begin{aligned} g_i(x_i) &= y_i, \quad i = 0 \text{ to } n-1 \\ g_i(x_{i+1}) &= y_{i+1}, \quad i = 0 \text{ to } n-1 \end{aligned}$$

The requirement that $g'(x)$ is continuous results in

$$g_i'(x_{i+1}) = g_{i+1}'(x_{i+1}), \quad i = 0 \text{ to } n-2$$

And the requirement that $g''(x)$ is continuous results in

$$g_i''(x_{i+1}) = g_{i+1}''(x_{i+1}), \quad i = 0 \text{ to } n-2$$

There are n cubic polynomials $g_i(x)$ and each cubic polynomial has four free coefficients; there are therefore a total of $4n$ unknown coefficients. The number of constraining equations from (5.1)-(5.4) is $2n + 2(n-1) = 4n - 2$. With $4n - 2$ constraints and $4n$ unknowns, two more conditions are required for a unique solution. These are usually chosen to be extra conditions on the first $g_0(x)$ and last $g_{n-1}(x)$ polynomials. We will discuss these extra conditions later.

We now proceed to determine equations for the unknown coefficients of the cubic polynomials. The polynomials and their first two derivatives are given by

$$\begin{aligned} g_i(x) &= a_i (x - x_i)^3 + b_i (x - x_i)^2 + c_i (x - x_i) + d_i \\ g_i'(x) &= 3a_i (x - x_i)^2 + 2b_i (x - x_i) + c_i \\ g_i''(x) &= 6a_i (x - x_i) + 2b_i \end{aligned}$$

We will consider the four conditions (5.1)-(5.4) in turn. From (5.1) and (5.5), we have

$$d_i = y_i, \quad i = 0 \text{ to } n-1,$$

which directly solves for all of the d-coefficients.

To satisfy (5.2), we first define

$$h_i = x_{i+1} - x_i$$

and

$$f_i = y_{i+1} - y_i$$

Now, from (5.2) and (5.5), using (5.8), we obtain the n equations

$$a_i h_i^3 + b_i h_i^2 + c_i h_i = f_i, \quad i = 0 \text{ to } n-1$$

From (5.3) and (5.6) we obtain the $n-1$ equations

$$3a_i h_i^2 + 2b_i h_i + c_i = c_{i+1}, \quad i = 0 \text{ to } n-2$$

From (5.4) and (5.7) we obtain the $n-1$ equations

$$3a_i h_i + b_i = b_{i+1} \quad i = 0 \text{ to } n-2$$

It is will be useful to include a definition of the coefficient b_n, which is now missing. (The index of the cubic polynomial coefficients only go up to $n-1$.) We simply extend (5.11) up to $i = n-1$ and so write

$$3a_{n-1} h_{n-1} + b_{n-1} = b_n$$

which can be viewed as a definition of b_n.

We now proceed to eliminate the sets of a - and c-coefficients (with the d-coefficients already eliminated in (5.8)) to find a system of linear equations for the b-coefficients. From (5.11) and (5.12), we can solve for the na-coefficients to find

$$a_i = \frac{1}{3h_i} (b_{i+1} - b_i), \quad i = 0 \text{ to } n-1$$

From (5.9), we can solve for the n c-coefficients as follows:

$$\begin{aligned} c_i &= \frac{1}{h_i} \left(f_i - a_i h_i^3 - b_i h_i^2 \right) \\ &= \frac{1}{h_i} \left(f_i - \frac{1}{3h_i} (b_{i+1} - b_i) h_i^3 - b_i h_i^2 \right) \\ &= \frac{f_i}{h_i} - \frac{1}{3} h_i (b_{i+1} + 2b_i), \quad i = 0 \text{ to } n-1. \end{aligned}$$

We can now find an equation for the b-coefficients by substituting (5.8), (5.13) and (5.14) into (5.10) :

$$\begin{aligned} 3 \left(\frac{1}{3h_i} (b_{i+1} - b_i) \right) h_i^2 + 2b_i h_i + \left(\frac{f_i}{h_i} - \frac{1}{3} h_i (b_{i+1} + 2b_i) \right) \\ = \left(\frac{f_{i+1}}{h_{i+1}} - \frac{1}{3} h_{i+1} (b_{i+2} + 2b_{i+1}) \right) \end{aligned}$$

which simplifies to

$$\frac{1}{3} h_i b_i + \frac{2}{3} (h_i + h_{i+1}) b_{i+1} + \frac{1}{3} h_{i+1} b_{i+2} = \frac{f_{i+1}}{h_{i+1}} - \frac{f_i}{h_i}$$

an equation that is valid for $i = 0$ to $n - 2$. Therefore, (5.15) represent $n - 1$ equations for the $n + 1$ unknown b-coefficients. Accordingly, we write the matrix equa- tion for the b-coefficients, leaving the first and last row absent, as

$$\begin{pmatrix} \cdots & \cdots & \cdots & \cdots & \text{missing} & \cdots & \cdots \\ \frac{1}{3}h_0 & \frac{2}{3}(h_0+h_1) & \frac{1}{3}h_1 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & \frac{1}{3}h_{n-2} & \frac{2}{3}(h_{n-2}+h_{n-1}) & \frac{1}{3}h_{n-1} \\ \cdots & \cdots & \cdots & \cdots & \text{missing} & \cdots & \cdots \end{pmatrix}$$

$$\begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \\ b_n \end{pmatrix} \begin{pmatrix} \text{missing} \\ \frac{f_1}{h_1} - \frac{f_0}{h_0} \\ \vdots \\ \frac{f_{n-1}}{h_{n-1}} - \frac{f_{n-2}}{h_{n-2}} \\ \text{missing} \end{pmatrix}$$

Once the missing first and last equations are specified, the matrix equation for the b-coefficients can be solved by Gaussian elimination. And once the b-coefficients are determined, the a - and c-coefficients can also be determined from (5.13) and (5.14), with the d-coefficients already known from (5.8). The piecewise cubic polynomials, then, are known and $g(x)$ can be used for interpolation to any value x satisfying $x_0 \le x \le x_n$

The missing first and last equations can be specified in several ways, and here we show the two ways that are allowed by the MATLAB function spline.m. The first way should be used when the derivative $g'(x)$ is known at the endpoints x_0 and x_n; that is, suppose we know the values of α and β such that

$$g_0'(x_0) = \alpha, \quad g_{n-1}'(x_n) = \beta.$$

From the known value of α, and using (5.6) and (5.14), we have

$$\begin{aligned} \alpha &= c_0 \\ &= \frac{f_0}{h_0} - \frac{1}{3}h_0(b_1 + 2b_0) \end{aligned}$$

Therefore, the missing first equation is determined to be

$$\frac{2}{3}h_0b_0 + \frac{1}{3}h_0b_1 = \frac{f_0}{h_0} - \alpha.$$

From the known value of β, and using $(5.6), (5.13)$, and (5.14), we have

$$\begin{aligned} \beta &= 3a_{n-1}h_{n-1}^2 + 2b_{n-1}h_{n-1} + c_{n-1} \\ &= 3\left(\frac{1}{3h_{n-1}}(b_n - b_{n-1})\right)h_{n-1}^2 + 2b_{n-1}h_{n-1} + \left(\frac{f_{n-1}}{h_{n-1}} - \frac{1}{3}h_{n-1}(b_n + 2b_{n-1})\right), \end{aligned}$$

which simplifies to

$$\frac{1}{3}h_{n-1}b_{n-1} + \frac{2}{3}h_{n-1}b_n = \beta - \frac{f_{n-1}}{h_{n-1}}$$

to be used as the missing last equation.

The second way of specifying the missing first and last equations is called the not- a-knot condition, which assumes that

$$g_0(x) = g_1(x), \quad g_{n-2}(x) = g_{n-1}(x)$$

Considering the first of these equations, from (5.5) we have

$$\begin{aligned} a_0(x - x_0)^3 + b_0(x - x_0)^2 + c_0(x - x_0) + d_0 \\ = a_1(x - x_1)^3 + b_1(x - x_1)^2 + c_1(x - x_1) + d_1 \end{aligned}$$

Now two cubic polynomials can be proven to be identical if at some value of x, the polynomials and their first three derivatives are identical. Our conditions of continuity at $x = x_1$ already require that at this value of x these two polynomials and their first two derivatives are identical. The polynomials themselves will be identical, then, if their third derivatives are also identical at $x = x_1$, or if

$$a_0 = a_1$$

From (5.13), we have

$$\frac{1}{3h_0}(b_1 - b_0) = \frac{1}{3h_1}(b_2 - b_1)$$

or after simplification

$$h_1 b_0 - (h_0 + h_1) b_1 + h_0 b_2 = 0$$

which provides us our missing first equation. A similar argument at $x = x_n - 1$ also provides us with our last equation,

$$h_{n-1} b_{n-2} - (h_{n-2} + h_{n-1}) b_{n-1} + h_{n-2} b_n = 0.$$

The MATLAB subroutines spline.m and ppval.m can be used for cubic spline interpolation (see also interp1.m). I will illustrate these routines in class and post sample code on the course web site.

Multidimensional interpolation

Suppose we are interpolating the value of a function of two variables,

$$z = f(x, y)$$

The known values are given by

$$z_{ij} = f(x_i, y_j)$$

with $i = 0, 1, \ldots, n$ and $j = 0, 1, \ldots, n$. Note that the (x, y) points at which $f(x, y)$ are known lie on a grid in the $x - y$ plane.

Let $z = g(x, y)$ be the interpolating function, satisfying $z_{ij} = g(x_i, y_j)$. A twodimensional interpolation to find the value of g at the point (x, y) may be done by first performing $n + 1$ one-dimensional interpolations in y to find the value of g at the $n + 1$ points $(x_0, y), (x_1, y), \ldots, (x_n, y)$, followed by a single one-dimensional interpolation in x to find the value of g at (x, y).

In other words, two-dimensional interpolation on a grid of dimension $(n + 1) \times (n + 1)$ is done by first performing $n + 1$ one-dimensional interpolations to the value y followed by a single one-dimensional interpolation to the value x. Twodimensional interpolation can be generalized to higher dimensions. The MATLAB functions to perform two-and three-dimensional interpolation are interp2.m and interp3.m. 5.4. MULTIDIMENSIONAL INTERPOLATION

Chapter 6

Integration

We want to construct numerical algorithms that can perform definite integrals of the form

$$I = \int_a^b f(x) dx$$

Calculating these definite integrals numerically is called numerical integration, numerical quadrature, or more simply quadrature.

Elementary formulas

We first consider integration from 0 to h, with h small, to serve as the building blocks for integration over larger domains. We here define I_h as the following integral:

$$I_h = \int_0^h f(x) dx.$$

To perform this integral, we consider a Taylor series expansion of $f(x)$ about the value $x = h/2$:

$$f(x) = f(h/2) + (x - h/2) f'(h/2) + \frac{(x - h/2)^2}{2} f''(h/2) + \frac{(x - h/2)^3}{6} f'''(h/2) + \frac{(x - h/2)^4}{24} f''''(h/2) + \ldots$$

Midpoint rule

The midpoint rule makes use of only the first term in the Taylor series expansion. Here, we will determine the error in this approximation. Integrating,

$$I_h = hf(h/2) + \int_0^h ((x-h/2)f'(h/2) + \frac{(x-h/2)^2}{2}f''(h/2) + \frac{(x-h/2)^3}{6}f'''(h/2) + \frac{(x-h/2)^4}{24}f''''(h/2) + \ldots\Big)\,dx.$$

Changing variables by letting $y = x - h/2$ and $dy = dx$, and simplifying the integral depending on whether the integrand is even or odd, we have

$$\begin{aligned} I_h =& hf(h/2) \\ &+ \int_{-h/2}^{h/2} \left(yf'(h/2) + \frac{y^2}{2}f''(h/2) + \frac{y^3}{6}f'''(h/2) + \frac{y^4}{24}f''''(h/2) + \ldots\right) dy \\ =& hf(h/2) + \int_0^{h/2} \left(y^2 f''(h/2) + \frac{y^4}{12}f''''(h/2) + \ldots\right) dy \end{aligned}$$

The integrals that we need here are

$$\int_0^{\frac{h}{2}} y^2 dy = \frac{h^3}{24}, \quad \int_0^{\frac{h}{2}} y^4 dy = \frac{h^5}{160}$$

Therefore,

$$I_h = hf(h/2) + \frac{h^3}{24}f''(h/2) + \frac{h^5}{1920}f''''(h/2) + \ldots$$

Trapezoidal rule

From the Taylor series expansion of $f(x)$ about $x = h/2$, we have

$$f(0) = f(h/2) - \frac{h}{2}f'(h/2) + \frac{h^2}{8}f''(h/2) - \frac{h^3}{48}f'''(h/2) + \frac{h^4}{384}f''''(h/2) + \ldots,$$

and

$$f(h) = f(h/2) + \frac{h}{2}f'(h/2) + \frac{h^2}{8}f''(h/2) + \frac{h^3}{48}f'''(h/2) + \frac{h^4}{384}f''''(h/2) + \ldots\ldots$$

Adding and multiplying by $h/2$ we obtain

$$\frac{h}{2}(f(0) + f(h)) = hf(h/2) + \frac{h^3}{8}f''(h/2) + \frac{h^5}{384}f''''(h/2) + \ldots.$$

We now substitute for the first term on the right-hand-side using the midpoint rule formula:

$$\begin{aligned} \tfrac{h}{2}(f(0) + f(h)) = &\left(I_h - \tfrac{h^3}{24}f''(h/2) - \tfrac{h^5}{1920}f''''(h/2)\right) \\ &+ \tfrac{h^3}{8}f''(h/2) + \tfrac{h^5}{384}f''''(h/2) + \ldots, \end{aligned}$$

and solving for I_h, we find

$$I_h = \frac{h}{2}(f(0) + f(h)) - \frac{h^3}{12}f''(h/2) - \frac{h^5}{480}f''''(h/2) + \ldots$$

Simpson's rule

To obtain Simpson's rule, we combine the midpoint and trapezoidal rule to eliminate the error term proportional to h^3. Multiplying (6.3) by two and adding to (6.4), we obtain

$$3I_h = h\left(2f(h/2) + \frac{1}{2}(f(0) + f(h))\right) + h^5\left(\frac{2}{1920} - \frac{1}{480}\right)f''''(h/2) + \ldots,$$

or

$$I_h = \frac{h}{6}\left(4f(h/2) + (f(0) + f(h))\right) - \frac{h^5}{2880}f''''(h/2) + \ldots$$

$$I_h = \frac{h}{6}(f(0) + 4f(h/2) + f(h)) - \frac{h^5}{2880} f''''(h/2) + \dots .$$

Usually, Simpson's rule is written by considering the three consecutive points $0, h$ and $2h$. Substituting $h \to 2h$, we obtain the standard result

$$I_{2h} = \frac{h}{3}(f(0) + 4f(h) + f(2h)) - \frac{h^5}{90} f''''(h) + \dots$$

Composite rules

We now use our elementary formulas obtained for (6.2) to perform the integral given by (6.1)

Trapezoidal rule

We suppose that the function $f(x)$ is known at the $n+1$ points labeled as $x_0, x_1, \dots, x_n$, with the endpoints given by $x_0 = a$ and $x_n = b$. Define

$$f_i = f(x_i), \quad h_i = x_{i+1} - x_i$$

Then the integral of (6.1) may be decomposed as

$$\begin{aligned} \int_a^b f(x)dx &= \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} f(x)dx \\ &= \sum_{i=0}^{n-1} \int_0^{h_i} f(x_i + s)\, ds \end{aligned}$$

where the last equality arises from the change-of-variables $s = x - x_i$. Applying the trapezoidal rule to the integral, we have

$$\int_a^b f(x)dx = \sum_{i=0}^{n-1} \frac{h_i}{2}(f_i + f_{i+1})$$

If the points are not evenly spaced, say because the data are experimental values, then the h_i may differ for each value of i and (6.6) is to be used directly.

However, if the points are evenly spaced, say because $f(x)$ can be computed, we have $h_i = h$, independent of i. We can then define

$$x_i = a + ih, \quad i = 0, 1, \dots, n$$

and since the end point b satisfies $b = a + nh$, we have

$$h = \frac{b-a}{n}$$

The composite trapezoidal rule for evenly space points then becomes

$$\begin{aligned} \int_a^b f(x)dx &= \frac{h}{2} \sum_{i=0}^{n-1} (f_i + f_{i+1}) \\ &= \frac{h}{2}(f_0 + 2f_1 + \cdots + 2f_{n-1} + f_n) \end{aligned}$$

The first and last terms have a multiple of one; all other terms have a multiple of two; and the entire sum is multiplied by $h/2$.

Simpson's rule

We here consider the composite Simpson's rule for evenly space points. We apply Simpson's rule over intervals of $2h$, starting from a and ending at b :

$$\begin{aligned} \int_a^b f(x)dx = \frac{h}{3}(f_0 + 4f_1 + f_2) + \frac{h}{3}(f_2 + 4f_3 + f_4) + \dots \\ + \frac{h}{3}(f_{n-2} + 4f_{n-1} + f_n) \end{aligned}$$

Note that n must be even for this scheme to work. Combining terms, we have

$$\int_a^b f(x)dx = \frac{h}{3}(f_0 + 4f_1 + 2f_2 + 4f_3 + 2f_4 + \cdots + 4f_{n-1} + f_n)$$

The first and last terms have a multiple of one; the even indexed terms have a multiple of 2; the odd indexed terms have a multiple of 4; and the entire sum is multiplied by $h/3$.

Local versus global error

Consider the elementary formula (6.4) for the trapezoidal rule, written in the form

$$\int_0^h f(x)dx = \frac{h}{2}(f(0) + f(h)) - \frac{h^3}{12}f''(\xi)$$

where ξ is some value satisfying $0 \leq \xi \leq h$, and we have used Taylor's theorem with the mean-value form of the remainder. We can also represent the remainder as

$$-\frac{h^3}{12}f''(\xi) = \mathrm{O}\left(h^3\right)$$

where $\mathrm{O}\left(h^3\right)$ signifies that when h is small, halving of the grid spacing h decreases the error in the elementary trapezoidal rule by a factor of eight. We call the terms represented by $\mathrm{O}\left(h^3\right)$ the Local Error.

More important is the Global Error which is obtained from the composite formula (6.7) for the trapezoidal rule. Putting in the remainder terms, we have

$$\int_a^b f(x)dx = \frac{h}{2}\left(f_0 + 2f_1 + \cdots + 2f_{n-1} + f_n\right) - \frac{h^3}{12}\sum_{i=0}^{n-1} f''\left(\xi_i\right)$$

where ξ_i are values satisfying $x_i \leq \xi_i \leq x_{i+1}$. The remainder can be rewritten as

$$-\frac{h^3}{12}\sum_{i=0}^{n-1} f''\left(\xi_i\right) = -\frac{nh^3}{12}\left\langle f''\left(\xi_i\right)\right\rangle$$

where $\left\langle f''\left(\xi_i\right)\right\rangle$ is the average value of all the $f''\left(\xi_i\right)'$ s. Now,

$$n = \frac{b-a}{h}$$

so that the error term becomes

$$\begin{aligned} -\frac{nh^3}{12}\left\langle f''\left(\xi_i\right)\right\rangle &= -\frac{(b-a)h^2}{12}\left\langle f''\left(\xi_i\right)\right\rangle \\ &= \mathrm{O}\left(h^2\right) \end{aligned}$$

Therefore, the global error is $\mathrm{O}\left(h^2\right)$. That is, a halving of the grid spacing only decreases the global error by a factor of four.

Similarly, Simpson's rule has a local error of $\mathrm{O}\left(h^5\right)$ and a global error of $\mathrm{O}\left(h^4\right)$.

Figure 6.1: Adaptive Simpson quadrature: Level 1.

Adaptive integration

The useful MATLAB function quad.m performs numerical integration using adaptive Simpson quadrature. The idea is to let the computation itself decide on the grid size required to achieve a certain level of accuracy. Moreover, the grid size need not be the same over the entire region of integration.

We begin the adaptive integration at what is called Level 1. The uniformly spaced points at which the function $f(x)$ is to be evaluated are shown in Fig. 6.1. The distance between the points a and b is taken to be $2h$, so that

$$h = \frac{b-a}{2}$$

Integration using Simpson's rule (6.5) with grid size h yields

$$I = \frac{h}{3}(f(a) + 4f(c) + f(b)) - \frac{h^5}{90}f''''(\xi)$$

where ξ is some value satisfying $a \leq \xi \leq b$.

Integration using Simpson's rule twice with grid size $h/2$ yields

$$I = \frac{h}{6}(f(a) + 4f(d) + 2f(c) + 4f(e) + f(b)) - \frac{(h/2)^5}{90} f''''(\xi_l) - \frac{(h/2)^5}{90} f''''(\xi_r)$$

with ξ_l and ξ_r some values satisfying $a \leq \xi_l \leq c$ and $c \leq \xi_r \leq b$.

We now define

$$\begin{aligned} S_1 &= \frac{h}{3}(f(a) + 4f(c) + f(b)) \\ S_2 &= \frac{h}{6}(f(a) + 4f(d) + 2f(c) + 4f(e) + f(b)), \\ E_1 &= -\frac{h^5}{90} f''''(\xi), \\ E_2 &= -\frac{h^5}{2^5 \cdot 90} \left(f''''(\xi_l) + f''''(\xi_r)\right). \end{aligned}$$

Now we ask whether S_2 is accurate enough, or must we further refine the calculation and go to Level 2? To answer this question, we make the simplifying approximation that all of the fourth-order derivatives of $f(x)$ in the error terms are equal; that is,

$$f''''(\xi) = f''''(\xi_l) = f''''(\xi_r) = C.$$

CHAPTER 6. INTEGRATION

Then

$$\begin{aligned} E_1 &= -\frac{h^5}{90} C \\ E_2 &= -\frac{h^5}{2^4 \cdot 90} C = \frac{1}{16} E_1 \end{aligned}$$

Then since

$$S_1 + E_1 = S_2 + E_2$$

and

$$E_1 = 16 E_2$$

we have for our estimate for the error term E_2,

$$E_2 = \frac{1}{15}(S_2 - S_1)$$

Therefore, given some specific value of the tolerance tol, if

$$\left|\frac{1}{15}(S_2 - S_1)\right| < \text{ tol}$$

then we can accept S_2 as I. If the tolerance is not achieved for I, then we proceed to Level 2.

The computation at Level 2 further divides the integration interval from a to b into the two integration intervals a to c and c to b, and proceeds with the above procedure independently on both halves. Integration can be stopped on either half provided the tolerance is less than tol/2 (since the sum of both errors must be less than tol). Otherwise, either half can proceed to Level 3, and so on.

As a side note, the two values of I given above (for integration with step size h and $h/2$) can be combined to give a more accurate value for I given by

$$I = \frac{16 S_2 - S_1}{15} + \mathrm{O}\left(h^7\right)$$

where the error terms of $\mathrm{O}\left(h^5\right)$ approximately cancel. This free lunch, so to speak, is called Richardson's extrapolation.

Chapter 7

Ordinary differential equations

We now discuss the numerical solution of ordinary differential equations. These include the initial value problem, the boundary value problem, and the eigenvalue problem. Before proceeding to the development of numerical methods, we review the analytical solution of some classic equations.

Examples of analytical solutions

Initial value problem

One classic initial value problem is the RC circuit. With R the resistor and C the capacitor, the differential equation for the charge q on the capacitor is given by

$$R\frac{dq}{dt} + \frac{q}{C} = 0.$$

If we consider the physical problem of a charged capacitor connected in a closed circuit to a resistor, then the initial condition is $q(0) = q_0$, where q_0 is the initial charge on the capacitor.

The differential equation (7.1) is separable, and separating and integrating from time $t = 0$ to t yields

$$\int_{q_0}^{q} \frac{dq}{q} = -\frac{1}{RC}\int_0^t dt$$

which can be integrated and solved for $q = q(t)$:

$$q(t) = q_0 e^{-t/RC}.$$

The classic second-order initial value problem is the RLC circuit, with differential equation

$$L\frac{d^2q}{dt^2} + R\frac{dq}{dt} + \frac{q}{C} = 0.$$

Here, a charged capacitor is connected to a closed circuit, and the initial conditions satisfy

$$q(0) = q_0, \quad \frac{dq}{dt}(0) = 0$$

The solution is obtained for the second-order equation by the ansatz

$$q(t) = e^{rt}$$

which results in the following so-called characteristic equation for r :

$$Lr^2 + Rr + \frac{1}{C} = 0$$

If the two solutions for r are distinct and real, then the two found exponential solutions can be multiplied by constants and added to form a general solution. The constants can then be determined by requiring the general solution to satisfy the two initial conditions. If the roots of the characteristic equation are complex or degenerate, a general solution to the differential equation can also be found.

Boundary value problems

The dimensionless equation for the temperature $y = y(x)$ along a linear heatconducting rod of length unity, and with an applied external heat source $f(x)$, is given by the differential equation

$$-\frac{d^2y}{dx^2} = f(x)$$

with $0 \le x \le 1$. Boundary conditions are usually prescribed at the end points of the rod, and here we assume that the temperature at both ends are maintained at zero so that

$$y(0) = 0, \quad y(1) = 0$$

The assignment of boundary conditions at two separate points is called a twopoint boundary value problem, in contrast to the initial value problem where the boundary conditions are prescribed at only a single point. Two-point boundary value problems typically

require a more sophisticated algorithm for a numerical solution than initial value problems.

Here, the solution of (7.2) can proceed by integration once $f(x)$ is specified. We assume that

$$f(x) = x(1-x)$$

so that the maximum of the heat source occurs in the center of the rod, and goes to zero at the ends.

The differential equation can then be written as

$$\frac{d^2y}{dx^2} = -x(1-x)$$

The first integration results in

$$\begin{aligned}\frac{dy}{dx} &= \int \left(x^2 - x\right) dx \\ &= \frac{x^3}{3} - \frac{x^2}{2} + c_1\end{aligned}$$

where c_1 is the first integration constant. Integrating again,

$$\begin{aligned}y(x) &= \int \left(\frac{x^3}{3} - \frac{x^2}{2} + c_1\right) dx \\ &= \frac{x^4}{12} - \frac{x^3}{6} + c_1 x + c_2\end{aligned}$$

where c_2 is the second integration constant. The two integration constants are determined by the boundary conditions. At $x = 0$, we have

$$0 = c_2$$

and at $x = 1$, we have

$$0 = \frac{1}{12} - \frac{1}{6} + c_1$$

so that $c_1 = 1/12$. Our solution is therefore

$$\begin{aligned}y(x) &= \frac{x^4}{12} - \frac{x^3}{6} + \frac{x}{12} \\ &= \frac{1}{12} x(1-x)\left(1 + x - x^2\right)\end{aligned}$$

The temperature of the rod is maximum at $x = 1/2$ and goes smoothly to zero at the ends.

Eigenvalue problem

The classic eigenvalue problem obtained by solving the wave equation by separation of variables is given by

$$\frac{d^2y}{dx^2} + \lambda^2 y = 0$$

with the two-point boundary conditions $y(0) = 0$ and $y(1) = 0$. Notice that $y(x) = 0$ satisfies both the differential equation and the boundary conditions. Other nonzero solutions for $y = y(x)$ are possible only for certain discrete values of λ. These values of λ are called the eigenvalues of the differential equation.

We proceed by first finding the general solution to the differential equation. It is easy to see that this solution is

$$y(x) = A\cos\lambda x + B\sin\lambda x$$

Imposing the first boundary condition at $x = 0$, we obtain

$$A = 0$$

The second boundary condition at $x = 1$ results in

$$B\sin\lambda = 0$$

Since we are searching for a solution where $y = y(x)$ is not identically zero, we must have

$$\lambda = \pi, 2\pi, 3\pi, ...$$

The corresponding negative values of λ are also solutions, but their inclusion only changes the corresponding values of the unknown B constant. A linear superposition of all the solutions results in the general solution

$$y(x) = \sum_{n=1}^{\infty} B_n \sin n\pi x.$$

For each eigenvalue $n\pi$, we say there is a corresponding eigenfunction $\sin n\pi x$. When the differential equation can not be solved analytically, a numerical method should be able to solve for both the eigenvalues and eigenfunctions.

Numerical methods: initial value problem

We begin with the simple Euler method, then discuss the more sophisticated RungeKutta methods, and conclude with the Runge-Kutta-Fehlberg method, as implemented in the MATLAB function ode45.m. Our differential equations are for $x = x(t)$, where the time t is the independent variable, and we will make use of the notation $\dot{x} = dx/dt$. This notation is still widely used by physicists and descends directly from the notation originally used by Newton.

Euler method

The Euler method is the most straightforward method to integrate a differential equation. Consider the first-order differential equation

$$\dot{x} = f(t, x),$$

with the initial condition $x(0) = x_0$. Define $t_n = n\Delta t$ and $x_n = x(t_n)$. A Taylor series expansion of x_{n+1} results in

$$\begin{aligned} x_{n+1} &= x(t_n + \Delta t) \\ &= x(t_n) + \Delta t \dot{x}(t_n) + \mathrm{O}(\Delta t^2) \\ &= x(t_n) + \Delta t f(t_n, x_n) + \mathrm{O}(\Delta t^2) \end{aligned}$$

The Euler Method is therefore written as

$$x_{n+1} = x(t_n) + \Delta t f(t_n, x_n)$$

We say that the Euler method steps forward in time using a time-step Δt, starting from the initial value $x_0 = x(0)$. The local error of the Euler Method is $\mathrm{O}(\Delta t^2)$. The global error, however, incurred when integrating to a time T, is a factor of $1/\Delta t$ larger and is given by $\mathrm{O}(\Delta t)$. It is therefore customary to call the Euler Method a first-order method.

Modified Euler method

This method is of a type that is called a predictor-corrector method. It is also the first of what are Runge-Kutta methods. As before, we want to solve (7.3). The idea is to average the value of $\dot{x}$ at the beginning and end of the time step. That is, we would like to modify the Euler method and write

$$x_{n+1} = x_n + \frac{1}{2}\Delta t \left(f(t_n, x_n) + f(t_n + \Delta t, x_{n+1})\right)$$

The obvious problem with this formula is that the unknown value x_{n+1} appears on the right-hand-side. We can, however, estimate this value, in what is called the predictor step. For the predictor step, we use the Euler method to find

$$x^p_{n+1} = x_n + \Delta t f(t_n, x_n)$$

The corrector step then becomes

$$x_{n+1} = x_n + \frac{1}{2}\Delta t \left(f(t_n, x_n) + f(t_n + \Delta t, x^p_{n+1})\right)$$

The Modified Euler Method can be rewritten in the following form that we will later identify as a Runge-Kutta method:

$$\begin{aligned} k_1 &= \Delta t f(t_n, x_n) \\ k_2 &= \Delta t f(t_n + \Delta t, x_n + k_1) \\ x_{n+1} &= x_n + \frac{1}{2}(k_1 + k_2) \end{aligned}$$

Second-order Runge-Kutta methods

We now derive all second-order Runge-Kutta methods. Higher-order methods can be similarly derived, but require substantially more algebra.

We consider the differential equation given by (7.3). A general second-order Runge-Kutta method may be written in the form

$$\begin{aligned} k_1 &= \Delta t f\left(t_n, x_n\right) \\ k_2 &= \Delta t f\left(t_n + \alpha \Delta t, x_n + \beta k_1\right) \\ x_{n+1} &= x_n + a k_1 + b k_2 \end{aligned}$$

with α, β, a and b constants that define the particular second-order Runge-Kutta method. These constants are to be constrained by setting the local error of the second-order Runge-Kutta method to be $\mathrm{O}\left(\Delta t^3\right)$. Intuitively, we might guess that two of the constraints will be $a + b = 1$ and $\alpha = \beta$.

We compute the Taylor series of x_{n+1} directly, and from the Runge-Kutta method, and require them to be the same to order Δt^2. First, we compute the Taylor series of x_{n+1}. We have

$$\begin{aligned} x_{n+1} &= x\left(t_n + \Delta t\right) \\ &= x\left(t_n\right) + \Delta t \dot{x}\left(t_n\right) + \frac{1}{2}(\Delta t)^2 \ddot{x}\left(t_n\right) + \mathrm{O}\left(\Delta t^3\right) \end{aligned}$$

Now,

$$\dot{x}\left(t_n\right) = f\left(t_n, x_n\right).$$

The second derivative is more complicated and requires partial derivatives. We have

$$\begin{aligned} \ddot{x}\left(t_n\right) &= \left.\frac{d}{dt} f(t, x(t))\right]_{t=t_n} \\ &= f_t\left(t_n, x_n\right) + \dot{x}\left(t_n\right) f_x\left(t_n, x_n\right) \\ &= f_t\left(t_n, x_n\right) + f\left(t_n, x_n\right) f_x\left(t_n, x_n\right) \end{aligned}$$

Therefore,

$$x_{n+1} = x_n + \Delta t f\left(t_n, x_n\right) + \frac{1}{2}(\Delta t)^2 \left(f_t\left(t_n, x_n\right) + f\left(t_n, x_n\right) f_x\left(t_n, x_n\right)\right)$$

Second, we compute x_{n+1} from the Runge-Kutta method given by (7.5). Substituting in k_1 and k_2, we have

$$x_{n+1} = x_n + a \Delta t f\left(t_n, x_n\right) + b \Delta t f\left(t_n + \alpha \Delta t, x_n + \beta \Delta t f\left(t_n, x_n\right)\right).$$

We Taylor series expand using

$$\begin{aligned} & f\left(t_n + \alpha \Delta t, x_n + \beta \Delta t f\left(t_n, x_n\right)\right) \\ &= f\left(t_n, x_n\right) + \alpha \Delta t f_t\left(t_n, x_n\right) + \beta \Delta t f\left(t_n, x_n\right) f_x\left(t_n, x_n\right) + \mathrm{O}\left(\Delta t^2\right) \end{aligned}$$

The Runge-Kutta formula is therefore

$$\begin{aligned} x_{n+1} = x_n &+ (a+b) \Delta t f\left(t_n, x_n\right) \\ &+ (\Delta t)^2 \left(\alpha b f_t\left(t_n, x_n\right) + \beta b f\left(t_n, x_n\right) f_x\left(t_n, x_n\right)\right) + \mathrm{O}\left(\Delta t^3\right) \end{aligned}$$

Comparing (7.6) and (7.7), we find

$$\begin{aligned} a + b &= 1 \\ \alpha b &= 1/2 \\ \beta b &= 1/2 \end{aligned}$$

There are three equations for four parameters, and there exists a family of secondorder Runge-Kutta methods.

The Modified Euler Method given by (7.4) corresponds to $\alpha = \beta = 1$ and $a = b = 1/2$. Another second-order Runge-Kutta method, called the Midpoint Method, corresponds to $\alpha = \beta = 1/2, a = 0$ and $b = 1$. This method is written as

$$\begin{aligned} k_1 &= \Delta t f\left(t_n, x_n\right) \\ k_2 &= \Delta t f\left(t_n + \frac{1}{2}\Delta t, x_n + \frac{1}{2} k_1\right) \\ x_{n+1} &= x_n + k_2 \end{aligned}$$

Higher-order Runge-Kutta methods

The general second-order Runge-Kutta method was given by (7.5). The general form of the third-order method is given by

$$\begin{aligned} k_1 &= \Delta t f\left(t_n, x_n\right) \\ k_2 &= \Delta t f\left(t_n + \alpha \Delta t, x_n + \beta k_1\right) \\ k_3 &= \Delta t f\left(t_n + \gamma \Delta t, x_n + \delta k_1 + \epsilon k_2\right) \\ x_{n+1} &= x_n + a k_1 + b k_2 + c k_3 \end{aligned}$$

The following constraints on the constants can be guessed: $\alpha = \beta, \gamma = \delta + \epsilon$, and $a + b + c = 1$. Remaining constraints need to be derived.

The fourth-order method has a k_1, k_2, k_3 and k_4. The fifth-order method requires up to k_6. The table below gives the order of the method and the number of stages required.

order	2	3	4	5	6	7	8
minimum # stages	2	3	4	6	7	9	11

Because of the jump in the number of stages required between the fourth-order and fifth-order method, the fourth-order Runge-Kutta method has some appeal. The general fourth-order method starts with 13 constants, and one then finds 11 constraints. A particularly simple fourth-order method that has been widely used is given by

$$\begin{aligned} k_1 &= \Delta t f\left(t_n, x_n\right) \\ k_2 &= \Delta t f\left(t_n + \frac{1}{2}\Delta t, x_n + \frac{1}{2}k_1\right) \\ k_3 &= \Delta t f\left(t_n + \frac{1}{2}\Delta t, x_n + \frac{1}{2}k_2\right) \\ k_4 &= \Delta t f\left(t_n + \Delta t, x_n + k_3\right) \\ x_{n+1} &= x_n + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \end{aligned}$$

Adaptive Runge-Kutta Methods

As in adaptive integration, it is useful to devise an ode integrator that automatically finds the appropriate Δt. The Dormand-Prince Method, which is implemented in MATLAB's ode45.m, finds the appropriate step size by comparing the results of a fifth-order and fourth-order method. It requires six function evaluations per time step, and constructs both a fifth-order and a fourth-order method from these function evaluations.

Suppose the fifth-order method finds x_{n+1} with local error $\mathrm{O}\left(\Delta t^6\right)$, and the fourth-order method finds x'_{n+1} with local error $\mathrm{O}\left(\Delta t^5\right)$. Let ε be the desired error tolerance of the method, and let e be the actual error. We can estimate e from the difference between the fifth-and fourth-order methods; that is,

$$e = \left|x_{n+1} - x'_{n+1}\right|$$

Now e is of $\mathrm{O}\left(\Delta t^5\right)$, where Δt is the step size taken. Let $\Delta\tau$ be the estimated step size required to get the desired error ε. Then we have

$$e/\varepsilon = (\Delta t)^5/(\Delta\tau)^5$$

or solving for $\Delta\tau$,

$$\Delta\tau = \Delta t\left(\frac{\varepsilon}{e}\right)^{1/5}$$

On the one hand, if $e < \varepsilon$, then we accept x_{n+1} and do the next time step using the larger value of $\Delta\tau$. On the other hand, if $e > \varepsilon$, then we reject the integration step and redo the time step using the smaller value of $\Delta\tau$. In practice, one usually increases the time step slightly less and decreases the time step slightly more to prevent the waste of too many failed time steps.

System of differential equations

Our numerical methods can be easily adapted to solve higher-order differential equations, or equivalently, a system of differential equations. First, we show how a second-order differential equation can be reduced to two first-order equations. Consider

$$\ddot{x} = f(t, x, \dot{x}).$$

This second-order equation can be rewritten as two first-order equations by defining $u = \dot{x}$. We then have the system

$$\begin{aligned} \dot{x} &= u, \\ \dot{u} &= f(t, x, u). \end{aligned}$$

This trick also works for higher-order equation. For another example, the thirdorder equation

$$\dddot{x} = f(t, x, \dot{x}, \ddot{x}),$$

can be written as

$$\begin{aligned} \dot{x} &= u \\ \dot{u} &= v \\ \dot{v} &= f(t, x, u, v). \end{aligned}$$

Now, we show how to generalize Runge-Kutta methods to a system of differential equations. As an example, consider the following system of two odes,

$$\begin{aligned} \dot{x} &= f(t, x, y), \\ \dot{y} &= g(t, x, y), \end{aligned}$$

with the initial conditions $x(0) = x_0$ and $y(0) = y_0$. The generalization of the commonly used fourth-order Runge-Kutta method would be

$$\begin{aligned} k_1 &= \Delta t f\left(t_n, x_n, y_n\right) \\ l_1 &= \Delta t g\left(t_n, x_n, y_n\right) \\ k_2 &= \Delta t f\left(t_n + \frac{1}{2}\Delta t, x_n + \frac{1}{2}k_1, y_n + \frac{1}{2}l_1\right) \\ l_2 &= \Delta t g\left(t_n + \frac{1}{2}\Delta t, x_n + \frac{1}{2}k_1, y_n + \frac{1}{2}l_1\right) \\ y_3 &= \Delta t f\left(t_n + \frac{1}{2}\Delta t, x_n + \frac{1}{2}k_2, y_n + \frac{1}{2}l_2\right) \\ l_3 &= \Delta t g\left(t_n + \frac{1}{2}\Delta t, x_n + \frac{1}{2}k_2, y_n + \frac{1}{2}l_2\right) \\ y_{n+1} &= y_n + \frac{1}{6}\left(l_1 + 2l_2 + 2l_3 + l_4\right) \\ k_4 &= \Delta t f\left(t_n + \Delta t, x_n + k_3, y_n + l_3\right) \\ l_4 &= \Delta t g\left(t_n + \Delta t, x_n + k_3, y_n + l_3\right) \\ &= x_n + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \\ &= \Delta \end{aligned}$$

Numerical methods: boundary value problem

Finite difference method

We consider first the differential equation

$$-\frac{d^2y}{dx^2} = f(x), \quad 0 \le x \le 1$$

with two-point boundary conditions

$$y(0) = A, \quad y(1) = B.$$

Equation (7.8) can be solved by quadrature, but here we will demonstrate a numerical solution using a finite difference method.

We begin by discussing how to numerically approximate derivatives. Consider the Taylor series approximation for $y(x + h)$ and $y(x - h)$, given by

$$\begin{aligned} y(x+h) &= y(x) + hy'(x) + \frac{1}{2}h^2y''(x) + \frac{1}{6}h^3y'''(x) + \frac{1}{24}h^4y''''(x) + \ldots, \\ y(x-h) &= y(x) - hy'(x) + \frac{1}{2}h^2y''(x) - \frac{1}{6}h^3y'''(x) + \frac{1}{24}h^4y''''(x) + \ldots \end{aligned}$$

The standard definitions of the derivatives give the first-order approximations

$$y'(x) = \frac{y(x+h) - y(x)}{h} + \mathrm{O}(h)$$
$$y'(x) = \frac{y(x) - y(x-h)}{h} + \mathrm{O}(h)$$

The more widely-used second-order approximation is called the central difference approximation and is given by

$$y'(x) = \frac{y(x+h) - y(x-h)}{2h} + \mathrm{O}\left(h^2\right)$$

The finite difference approximation to the second derivative can be found from considering

$$y(x+h) + y(x-h) = 2y(x) + h^2 y''(x) + \frac{1}{12}h^4 y''''(x) + \ldots,$$

from which we find

$$y''(x) = \frac{y(x+h) - 2y(x) + y(x-h)}{h^2} + \mathrm{O}\left(h^2\right).$$

Sometimes a second-order method is required for x on the boundaries of the domain. For a boundary point on the left, a second-order forward difference method requires the additional Taylor series

$$y(x+2h) = y(x) + 2hy'(x) + 2h^2 y''(x) + \frac{4}{3}h^3 y'''(x) + \ldots$$

We combine the Taylor series for $y(x+h)$ and $y(x+2h)$ to eliminate the term proportional to h^2 :

$$y(x+2h) - 4y(x+h) = -3y(x) - 2hy'(x) + \mathrm{O}\left(\mathrm{h}^3\right)$$

Therefore,

$$y'(x) = \frac{-3y(x) + 4y(x+h) - y(x+2h)}{2h} + \mathrm{O}\left(h^2\right)$$

For a boundary point on the right, we send $h \to -h$ to find

$$y'(x) = \frac{3y(x) - 4y(x-h) + y(x-2h)}{2h} + \mathrm{O}\left(h^2\right)$$

We now write a finite difference scheme to solve (7.8). We discretize x by defining $x_i = ih, i = 0, 1, \ldots, n+1$. Since $x_{n+1} = 1$, we have $h = 1/(n+1)$. The functions $y(x)$ and $f(x)$ are discretized as $y_i = y(x_i)$ and $f_i = f(x_i)$. The second-order differential equation (7.8) then becomes for the interior points of the domain

$$-y_{i-1} + 2y_i - y_{i+1} = h^2 f_i, \quad i = 1, 2, \ldots n,$$

with the boundary conditions $y_0 = A$ and $y_{n+1} = B$. We therefore have a linear system of equations to solve. The first and nth equation contain the boundary conditions and are given by

$$2y_1 - y_2 = h^2 f_1 + A$$
$$-y_{n-1} + 2y_n = h^2 f_n + B$$

The second and third equations, etc., are

$$-y_1 + 2y_2 - y_3 = h^2 f_2$$
$$-y_2 + 2y_3 - y_4 = h^2 f_3$$

In matrix form, we have

$$\begin{pmatrix} 2 & -1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & \cdots & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \\ y_n \end{pmatrix} = \begin{pmatrix} h^2 f_1 + A \\ h^2 f_2 \\ h^2 f_3 \\ \vdots \\ h^2 f_{n-1} \\ h^2 f_n + B \end{pmatrix}$$

The matrix is tridiagonal, and a numerical solution by Guassian elimination can be done quickly. The matrix itself is easily constructed using the MATLAB function diag.m and ones.m. As excerpted from the MATLAB help page, the function call ones (m, n) returns an m-by-n matrix of ones, and the function call $\mathrm{diag}(\mathrm{v}, \mathrm{k})$, when v is a vector with n components, is a square matrix of order $\mathrm{n} + \mathrm{abs}(\mathrm{k})$ with the elements of v on the k-th diagonal: $k = 0$ is the main diagonal, $k > 0$ is above the main diagonal and $k < 0$ is below the main diagonal. The $n \times n$ matrix above can be constructed by the MATLAB code

$$M = \mathrm{diag}(-\ \mathrm{ones}\ (n-1,1), -1) + \mathrm{diag}\,(2^*\ \mathrm{ones}\ (n,1), 0) + \mathrm{diag}(-\ \mathrm{ones}\ (n-1,1), 1);$$

The right-hand-side, provided f is a given n-by-1 vector, can be constructed by the MATLAB code

$$\mathrm{b} = \mathrm{h}\text{^}2^*\mathrm{f}; \mathrm{b}(1) = \mathrm{b}(1) + \mathrm{A}; \mathrm{b}(\mathrm{n}) = \mathrm{b}(\mathrm{n}) + \mathrm{B}$$

and the solution for u is given by the MATLAB code

$$\mathrm{y} = \mathrm{M}\backslash\mathrm{b}$$

Shooting method

The finite difference method can solve linear odes. For a general ode of the form

$$\frac{d^2y}{dx^2} = f(x, y, dy/dx)$$

with $y(0) = A$ and $y(1) = B$, we use a shooting method. First, we formulate the ode as an initial value problem. We have

$$\begin{aligned} \frac{dy}{dx} &= z \\ \frac{dz}{dx} &= f(x, y, z) \end{aligned}$$

The initial condition $y(0) = A$ is known, but the second initial condition $z(0) = b$ is unknown. Our goal is to determine b such that $y(1) = B$.

In fact, this is a root-finding problem for an appropriately defined function. We define the function $F = F(b)$ such that

$$F(b) = y(1) - B$$

In other words, $F(b)$ is the difference between the value of $y(1)$ obtained from integrating the differential equations using the initial condition $z(0) = b$, and B. Our root-finding routine will want to solve $F(b) = 0$. (The method is called shooting because the slope of the solution curve for $y = y(x)$ at $x = 0$ is given by b, and the solution hits the value $y(1)$ at $x = 1$. This looks like pointing a gun and trying to shoot the target, which is B.)

To determine the value of b that solves $F(b) = 0$, we iterate using the Secant method, given by

$$b_{n+1} = b_n - F\,(b_n)\,\frac{b_n - b_{n-1}}{F\,(b_n) - F\,(b_{n-1})}$$

We need to start with two initial guesses for b, solving the ode for the two corresponding values of $y(1)$. Then the Secant Method will give us the next value of b to try, and we iterate until $|y(1) - B| <$ tol, where tol is some specified tolerance for the error

Numerical methods: eigenvalue problem

For illustrative purposes, we develop our numerical methods for what is perhaps the simplest eigenvalue ode. With $y = y(x)$ and $0 \leq x \leq 1$, this simple ode is given by

$$y'' + \lambda^2 y = 0$$

To solve (7.9) numerically, we will develop both a finite difference method and a shooting method. Furthermore, we will show how to solve (7.9) with homogeneous boundary conditions on either the function y or its derivative y'.

Finite difference method

We first consider solving (7.9) with the homogeneous boundary conditions $y(0) = y(1) = 0$. In this case, we have already shown that the eigenvalues of (7.9) are given by $\lambda = \pi, 2\pi, 3\pi, \ldots$.

With n interior points, we have $x_i = ih$ for $i = 0, \dots, n+1$, and $h = 1/(n+1)$. Using the centered-finite-difference approximation for the second derivative, (7.9) becomes

$$y_{i-1} - 2y_i + y_{i+1} = -h^2\lambda^2 y_i$$

Applying the boundary conditions $y_0 = y_{n+1} = 0$, the first equation with $i = 1$, and the last equation with $i = n$, are given by

$$-2y_1 + y_2 = -h^2\lambda^2 y_1$$
$$y_{n-1} - 2y_n = -h^2\lambda^2 y_n$$

The remaining $n-2$ equations are given by (7.10) for $i = 2, \dots, n-1$ It is of interest to see how the solution develops with increasing n. The smallest possible value is $n = 1$, corresponding to a single interior point, and since $h = 1/2$ we have

$$-2y_1 = -\frac{1}{4}\lambda^2 y_1$$

so that $\lambda^2 = 8$, or $\lambda = 2\sqrt{2} = 2.8284$. This is to be compared to the first eigenvalue which is $\lambda = \pi$. When $n = 2$, we have $h = 1/3$, and the resulting two equations written in matrix form are given by

$$\begin{pmatrix} -2 & 1 \\ 1 & -2 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = -\frac{1}{9}\lambda^2 \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$

This is a matrix eigenvalue problem with the eigenvalue given by $\mu = -\lambda^2/9$. The solution for μ is arrived at by solving

$$\det \begin{pmatrix} -2-\mu & 1 \\ 1 & -2-\mu \end{pmatrix} = 0$$

with resulting quadratic equation

$$(2+\mu)^2 - 1 = 0$$

The solutions are $\mu = -1, -3$, and since $\lambda = 3\sqrt{-\mu}$, we have $\lambda = 3, 3\sqrt{3} = 5.1962$. These two eigenvalues serve as rough approximations to the first two eigenvalues π and 2π.

With A an n-by-n matrix, the MATLAB variable mu=eig(A) is a vector containing the n eigenvalues of the matrix A. The built-in function eig.m can therefore be used to find the eigenvalues. With n grid points, the smaller eigenvalues will converge more rapidly than the larger ones.

We can also consider boundary conditions on the derivative, or mixed boundary conditions. For example, consider the mixed boundary conditions given by $y(0) = 0$ and $y'(1) = 0$. The eigenvalues of (7.9) can then be determined analytically to be $\lambda_i = (i - 1/2)\pi$, with i a natural number.

The difficulty we now face is how to implement a boundary condition on the derivative. Our computation of y'' uses a second-order method, and we would like the computation of the first derivative to also be second order. The condition $y'(1) = 0$ occurs on the right-most boundary, and we can make use of the secondorder backward-difference approximation to the derivative that we have previously derived. This finite-difference approximation for $y'(1)$ can be written as

$$y'_{n+1} = \frac{3y_{n+1} - 4y_n + y_{n-1}}{2h}.$$

Now, the nth finite-difference equation was given by

$$y_{n-1} - 2y_n + y_{n+1} = -h^2 y_n,$$

and we now replace the value y_{n+1} using (7.11); that is,

$$y_{n+1} = \frac{1}{3}\left(2hy'_{n+1} + 4y_n - y_{n-1}\right).$$

Implementing the boundary condition $y'_{n+1} = 0$, we have

$$y_{n+1} = \frac{4}{3}y_n - \frac{1}{3}y_{n-1}$$

Therefore, the nth finite-difference equation becomes

$$2 \qquad 2$$

$$\frac{\tilde{}}{3}y_{n-1} - \frac{\tilde{}}{3}y_n = -h^2\lambda^2 y_n$$

For example, when $n = 2$, the finite difference equations become

$$\begin{pmatrix} -2 & 1 \\ \frac{2}{3} & -\frac{2}{3} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = -\frac{1}{9}\lambda^2 \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$

The eigenvalues of the matrix are now the solution of

$$(\mu + 2)\left(\mu + \frac{2}{3}\right) - \frac{2}{3} = 0$$

Or

$$3\mu^2 + 8\mu + 2 = 0$$

Therefore, $\mu = (-4 \pm \sqrt{10})/3$, and we find $\lambda = 1.5853, 4.6354$, which are approximations to $\pi/2$ and $3\pi/2$.

Shooting method

We apply the shooting method to solve (7.9) with boundary conditions $y(0) = y(1) = 0$. The initial value problem to solve is

$$\begin{aligned} y' &= z \\ z' &= -\lambda^2 y \end{aligned}$$

with known boundary condition $y(0) = 0$ and an unknown boundary condition on $y'(0)$. In fact, any nonzero boundary condition on $y'(0)$ can be chosen: the differential equation is linear and the boundary conditions are homogeneous, so that if $y(x)$ is an eigenfunction then so is $Ay(x)$. What we need to find here is the value of λ such that $y(1) = 0$. In other words, choosing $y'(0) = 1$, we solve

$$F(\lambda) = 0$$

where $F(\lambda) = y(1)$, obtained by solving the initial value problem. Again, an iteration for the roots of $F(\lambda)$ can be done using the Secant Method. For the eigenvalue problem, there are an infinite number of roots, and the choice of the two initial guesses for λ will then determine to which root the iteration will converge.

For this simple problem, it is possible to write explicitly the equation $F(\lambda) = 0$. The general solution to (7.9) is given by

$$y(x) = A\cos(\lambda x) + B\sin(\lambda x).$$

The initial condition $y(0) = 0$ yields $A = 0$. The initial condition $y'(0) = 1$ yields

$$B = 1/\lambda$$

Therefore, the solution to the initial value problem is

$$y(x) = \frac{\sin(\lambda x)}{\lambda}$$

The function $F(\lambda) = y(1)$ is therefore given by

$$F(\lambda) = \frac{\sin\lambda}{\lambda}$$

and the roots occur when $\lambda = \pi, 2\pi, \ldots$.

If the boundary conditions were $y(0) = 0$ and $y'(1) = 0$, for example, then we would simply redefine $F(\lambda) = y'(1)$. We would then have

$$F(\lambda) = \frac{\cos\lambda}{\lambda}$$

and the roots occur when $\lambda = \pi/2, 3\pi/2, \ldots$

CHAPTER OVERVIEW

1: IEEE Arithmetic

1.1: Definitions
1.2: Numbers with a Decimal or Binary Point
1.3: Examples of Binary Numbers
1.4: Hex Numbers
1.5: 4-bit Unsigned Integers as Hex Numbers
1.6: IEEE Single Precision Format
1.7: Special Numbers
1.8: Examples of Computer Numbers
1.9: Inexact Numbers
1.10: Machine Epsilon
1.11: IEEE Double Precision Format
1.12: Roundoff Error Example

This page titled 1: IEEE Arithmetic is shared under a CC BY 1.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov.

1.1: Definitions

Bit	= 0 or 1	
Byte	= 8 bits	
Word	= Reals:	4 bytes (single precision)
		8 bytes (double precision)
	= Integers:	$1, 2, 4,$ or 8 byte signed
		$1, 2, 4,$ or 8 byte unsigned

This page titled 1.1: Definitions is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

1.2: Numbers with a Decimal or Binary Point

	□	□	□	□	□	□	□	□	□
Decimal:	10^3	10^2	10^1	10^0		10^{-1}	10^{-2}	10^{-3}	10^{-4}
Binary:	2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}	2^{-4}

This page titled 1.2: Numbers with a Decimal or Binary Point is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

1.3: Examples of Binary Numbers

Decimal	Binary
1	1
2	10
3	11
4	100
0.5	0.1
1.5	1.1

This page titled 1.3: Examples of Binary Numbers is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

1.4: Hex Numbers

$$\{0,1,2,3,\ldots,9,10,11,12,13,14,15\} = \{0,1,2,3\ldots\ldots..9,\mathrm{a},\mathrm{b},\mathrm{c},\mathrm{d},\mathrm{e},\mathrm{f}\}$$

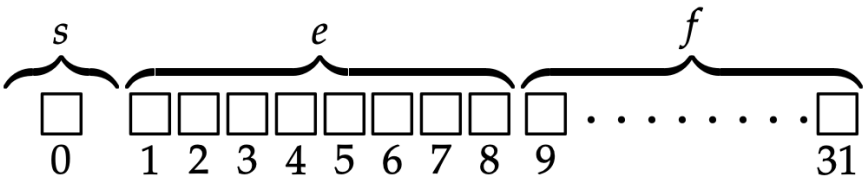
This page titled 1.4: Hex Numbers is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

1.5: 4-bit Unsigned Integers as Hex Numbers

Decimal	Binary	Hex
1	0001	1
2	0010	2
3	0011	3
⋮	⋮	⋮
10	1010	a
⋮	⋮	⋮
15	1111	f

This page titled 1.5: 4-bit Unsigned Integers as Hex Numbers is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

1.6: IEEE Single Precision Format



Misplaced '#'

$$\begin{aligned} \text{where} \quad & \text{s} = \text{sign} \\ & \text{e} = \text{biased exponent} \\ & \text{p} = \text{e-127} = \text{exponent} \\ & \text{1.f} = \text{signigicand (use binary point)} \end{aligned} \tag{1.6.1}$$

This page titled 1.6: IEEE Single Precision Format is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

1.7: Special Numbers

$$\text{Smallest exponent:} \quad \text{e} = 00000000, \text{ represents denormal numbers}(1.f \rightarrow 0.f) \tag{1.7.1}$$

$$\begin{aligned} \text{Largest exponent:} \quad & \text{e} = 11111111, \text{ represents } \pm\infty, \text{ if f} = 0 \\ & \text{e} = 11111111, \text{ represents NaN, if f} \neq 0 \end{aligned} \tag{1.7.2}$$

$$\begin{aligned} \text{Number Range:} \quad & \text{e} = 11111111 = 2^8 - 1 = && \text{reserved} \\ & \text{e} = 00000000 = 0 && \text{reserved} \\ \text{so,} \quad & \text{p} = \text{e} - 127 \text{ is} \\ & 1 - 127 \leq \text{p} \leq 254 - 127 \\ & -126 \leq \text{p} \leq 127 \end{aligned} \tag{1.7.3}$$

$$\begin{aligned} \text{Smallest positive normal number} \\ &= 1.00000000 \cdots \cdot .0000 \times 2^{-126} \\ &\simeq 1.2 \times 10^{-38} \\ &\text{bin: 00000000100000000000000000000000} \\ &\text{hex: 00800000} \\ &\text{MATLAB: realmin('single')} \end{aligned} \tag{1.7.4}$$

$$\begin{aligned} \text{Largest positive number} \\ &= 1.11111111 \cdots \cdots \cdot 1111 \times 2^{127} \\ &= (1 + (1 - 2^{-23})) \times 2^{127} \\ &\simeq 2^{128} \simeq 3.4 \times 10^{38} \\ &\text{bin: 01111111011111111111111111111111} \\ &\text{hex: 7f7fffff} \\ &\text{MATLAB: realmax('single')} \end{aligned} \tag{1.7.5}$$

$$\begin{aligned} \text{Zero} \\ &\text{bin: 0000 0000 0000 0000 0000 0000 0000 0000} \\ &\text{hex: 00000000} \end{aligned} \tag{1.7.6}$$

$$\begin{aligned} \text{Subnormal numbers} \\ &\text{Allow 1.f} \rightarrow \text{ (in software)} \\ &\text{Smallest positive number } = 0.00000000 \cdots \cdots 0001 \times 2^{-126} \\ &= 2^{-23} \times 2^{-126} \simeq 1.4 \times 10^{-45} \end{aligned} \tag{1.7.7}$$

This page titled 1.7: Special Numbers is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

1.8: Examples of Computer Numbers

What is $1.0, 2.0 \& 1/2$ in hex?

$1.0 = (-1)^0 \times 2^{(127-127)} \times 1.0$

Therefore, $s = 0, e = 01111111, f = 00000000000000000000000$

bin: 00111111100000000000000000000000

hex: 3 f 800000

$2.0 = (-1)^0 \times 2^{(128-127)} \times 1.0$

Therefore, $s = 0, e = 10000000, f = 00000000000000000000000$

bin: 01000000010000000000000000000000

hex: 40000000

$1/2 = (-1)^0 \times 2^{(126-127)} \times 1.0$

Therefore, $s = 0, e = 01111110, f = 00000000000000000000000$

bin: 00111111000000000000000000000000

hex: 3f00 0000

This page titled 1.8: Examples of Computer Numbers is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

1.9: Inexact Numbers

Example:

$$\frac{1}{3} = (-1)^0 \times \frac{1}{4} \times \left(1 + \frac{1}{3}\right),$$

so that $p = e - 127 = -2$ and $e = 125 = 128 - 3$, or in binary, $e = 01111101$. How is $f = 1/3$ represented in binary? To compute binary number, multiply successively by 2 as follows:

$0.333\ldots$	$0.$
$0.666\ldots$	0.0
$1.333\ldots$	0.01
$0.666\ldots$	0.010
$1.333\ldots$	0.0101
etc.	

so that $1/3$ exactly in binary is $0.010101\ldots$ With only 23 bits to represent f, the number is inexact and we have

$$f = 01010101010101010101011$$

where we have rounded to the nearest binary number (here, rounded up). The machine number $1/3$ is then represented as

$$00111110101010101010101010101011$$

or in hex

$$3eaaaaab.$$

Find smallest positive integer that is not exact in single pre- cision

Let N be the smallest positive integer that is not exact. Now, I claim that

$$N - 2 = 2^{23} \times 1.11\ldots1$$

and

$$N - 1 = 2^{24} \times 1.00\ldots0$$

The integer N would then require a one-bit in the 2^{-24} position, which is not available. Therefore, the smallest positive integer that is not exact is $2^{24} + 1 = 16777217$. In MATLAB, single (2^{24}) has the same value as single $(2^{24} + 1)$. Since single $(2^{24} + 1)$ is exactly halfway between the two consecutive machine numbers 2^{24} and $2^{24} + 2$, MATLAB rounds to the number with a final zero-bit in f, which is 2^{24}.

This page titled 1.9: Inexact Numbers is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

1.10: Machine Epsilon

Machine epsilon (ϵ_{mach}) is the distance between 1 and the next largest number. If $0 \leq \delta < \epsilon_{\text{mach}}/2$, then $1 + \delta = 1$ in computer math. Also since

$$x + y = x(1 + y/x)$$

if $0 \leq y/x < \epsilon_{\text{mach}}/2$, then $x + y = x$ in computer math.

Find ϵ_{mach}

The number 1 in the IEEE format is written as

$$1 = 2^0 \times 1.000 \ldots 0,$$

with $230'$ s following the binary point. The number just larger than 1 has a 1 in the 23 rd position after the decimal point. Therefore,

$$\epsilon_{\text{mach}} = 2^{-23} \approx 1.192 \times 10^{-7}$$

What is the distance between 1 and the number just smaller than 1? Here, the number just smaller than one can be written as

$$2^{-1} \times 1.111 \ldots 1 = 2^{-1}\left(1 + \left(1 - 2^{-23}\right)\right) = 1 - 2^{-24}$$

Therefore, this distance is $2^{-24} = \epsilon_{\text{mach}}/2$.

The spacing between numbers is uniform between powers of 2, with logarithmic spacing of the powers of 2 . That is, the spacing of numbers between 1 and 2 is 2^{-23}, between 2 and 4 is 2^{-22}, between 4 and 8 is 2^{-21}, etc. This spacing changes for denormal numbers, where the spacing is uniform all the way down to zero.

Find the machine number just greater than 5

A rough estimate would be $5\left(1 + \epsilon_{\text{mach}}\right) = 5 + 5\epsilon_{\text{mach}}$, but this is not exact. The exact answer can be found by writing

$$5 = 2^2\left(1 + \frac{1}{4}\right)$$

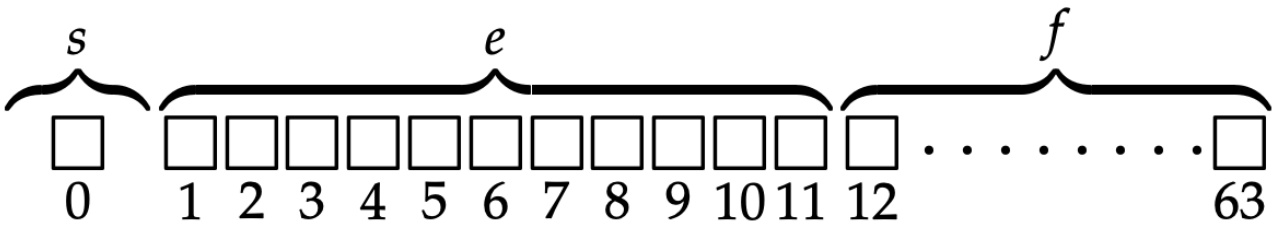
so that the next largest number is

$$2^2\left(1 + \frac{1}{4} + 2^{-23}\right) = 5 + 2^{-21} = 5 + 4\epsilon_{\text{mach}}$$

This page titled 1.10: Machine Epsilon is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

1.11: IEEE Double Precision Format

Most computations take place in double precision, where round-off error is reduced, and all of the above calculations in single precision can be repeated for double precision. The format is



$$\# = (-1)^s \times 2^{e-1023} \times 1.\text{f}$$

where
s = sign
e = biased exponent
p=e-1023 = exponent
1.f = significand (use binary point)

(1.11.1)

This page titled 1.11: IEEE Double Precision Format is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

1.12: Roundoff Error Example

Consider solving the quadratic equation

$$x^2 + 2bx - 1 = 0$$

where b is a parameter. The quadratic formula yields the two solutions

$$x_\pm = -b \pm \sqrt{b^2 + 1}$$

Consider the solution with $b > 0$ and $x > 0$ (the x_+solution) given by

$$x = -b + \sqrt{b^2 + 1}$$

As $b \to \infty$

$$\begin{aligned} x &= -b + \sqrt{b^2 + 1} \\ &= -b + b\sqrt{1 + 1/b^2} \\ &= b\left(\sqrt{1 + 1/b^2} - 1\right) \\ &\approx b\left(1 + \frac{1}{2b^2} - 1\right) \\ &= \frac{1}{2b}. \end{aligned}$$

Now in double precision, realmin $\approx 2.2 \times 10^{-308}$ and we would like x to be accurate to this value before it goes to 0 via denormal numbers. Therefore, x should be computed accurately to $b \approx 1/(2 \times$ realmin $) \approx 2 \times 10^{307}$. What happens if we compute (1.1) directly? Then $x = 0$ when $b^2 + 1 = b^2$, or $1 + 1/b^2 = 1$. That is $1/b^2 = \epsilon_{\text{mach}}/2$, or $b = \sqrt{2}/\sqrt{\epsilon_{\text{mach}}} \approx 10^8$. For a subroutine written to compute the solution of a quadratic for a general user, this is not good enough. The way for a software designer to solve this problem is to compute the solution for x as

$$x = \frac{1}{b\left(1 + \sqrt{1 + 1/b^2}\right)}$$

In this form, if $1 + 1/b^2 = 1$, then $x = 1/2b$ which is the correct asymptotic form.

This page titled 1.12: Roundoff Error Example is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

2: Root Finding

Solve $f(x) = 0$ for x, when an explicit analytical solution is impossible.

2: Root Finding is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

2.1: Bisection Method

The bisection method is the easiest to numerically implement and almost always works. The main disadvantage is that convergence is slow. If the bisection method results in a computer program that runs too slow, then other faster methods may be chosen; otherwise it is a good choice of method.

We want to construct a sequence $x_0, x_1, x_2, ...$ that converges to the root $x = r$ that solves $f(x) = 0$. We choose x_0 and x_1 such that $x_0 < r < x_1$. We say that x_0 and x_1 bracket the root. With $f(r) = 0$, we want $f(x_0)$ and $f(x_1)$ to be of opposite sign, so that $f(x_0)f(x_1) < 0$. We then assign x_2 to be the midpoint of x_0 and x_1, that is $x_2 = (x_0 + x_1)/2$, or

$$x_2 = x_0 + \frac{x_1 - x_0}{2}.$$

The sign of $f(x_2)$ can then be determined. The value of x_3 is then chosen as either the midpoint of x_0 and x_2 or as the midpoint of x_2 and x_1, depending on whether x_0 and x_2 bracket the root, or x_2 and x_1 bracket the root. The root, therefore, stays bracketed at all times. The algorithm proceeds in this fashion and is typically stopped when the increment to the left side of the bracket (above, given by $(x_1 - x_0)/2$) is smaller than some required precision.

This page titled 2.1: Bisection Method is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

2.2: Newton's Method

This is the fastest method, but requires analytical computation of the derivative of $f(x)$. Also, the method may not always converge to the desired root.

We can derive Newton's Method graphically, or by a Taylor series. We again want to construct a sequence $x_0, x_1, x_2, ...$ that converges to the root $x = r$. Consider the x_{n+1} member of this sequence, and Taylor series expand $f(x_{n+1})$ about the point x_n. We have

$$f(x_{n+1}) = f(x_n) + (x_{n+1} - x_n) f'(x_n) + ...$$

To determine x_{n+1}, we drop the higher-order terms in the Taylor series, and assume $f(x_{n+1}) = 0$. Solving for x_{n+1}, we have

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Starting Newton's Method requires a guess for x_0, hopefully close to the root $x = r$.

This page titled 2.2: Newton's Method is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

2.3: Secant Method

The Secant Method is second best to Newton's Method, and is used when a faster convergence than Bisection is desired, but it is too difficult or impossible to take an analytical derivative of the function $f(x)$. We write in place of $f'(x_n)$,

$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$$

Starting the Secant Method requires a guess for both x_0 and x_1.

2.3.1. Estimate $\sqrt{2} = 1.41421356$ using Newton's Method

The $\sqrt{2}$ is the zero of the function $f(x) = x^2 - 2$. To implement Newton's Method, we use $f'(x) = 2x$. Therefore, Newton's Method is the iteration

$$x_{n+1} = x_n - \frac{x_n^2 - 2}{2x_n}$$

We take as our initial guess $x_0 = 1$. Then

$$\begin{aligned} x_1 &= 1 - \frac{-1}{2} = \frac{3}{2} = 1.5 \\ x_2 &= \frac{3}{2} - \frac{\frac{9}{4} - 2}{3} = \frac{17}{12} = 1.416667, \\ x_3 &= \frac{17}{12} - \frac{\frac{17^2}{12^2} - 2}{\frac{17}{6}} = \frac{577}{408} = 1.41426. \end{aligned}$$

2.3.2. Example of fractals using Newton's Method

Consider the complex roots of the equation $f(z) = 0$, where

$$f(z) = z^3 - 1$$

These roots are the three cubic roots of unity. With

$$e^{i2\pi n} = 1, \quad n = 0, 1, 2, ...$$

the three unique cubic roots of unity are given by

$$1, \quad e^{i2\pi/3}, \quad e^{i4\pi/3}$$

With

$$e^{i\theta} = \cos\theta + i\sin\theta,$$

and $\cos(2\pi/3) = -1/2, \sin(2\pi/3) = \sqrt{3}/2$, the three cubic roots of unity are

$$r_1 = 1, \quad r_2 = -\frac{1}{2} + \frac{\sqrt{3}}{2}i, \quad r_3 = -\frac{1}{2} - \frac{\sqrt{3}}{2}i$$

The interesting idea here is to determine which initial values of z_0 in the complex plane converge to which of the three cubic roots of unity.

Newton's method is

$$z_{n+1} = z_n - \frac{z_n^3 - 1}{3z_n^2}$$

If the iteration converges to r_1, we color z_0 red; r_2, blue; r_3, green. The result will be shown in lecture.

This page titled 2.3: Secant Method is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

2.4: Order of Convergence

Let r be the root and x_n be the nth approximation to the root. Define the error as

$$\epsilon_n = r - x_n$$

If for large n we have the approximate relationship

$$|\epsilon_{n+1}| = k\,|\epsilon_n|^p\,,$$

with k a positive constant, then we say the root-finding numerical method is of order p. Larger values of p correspond to faster convergence to the root. The order of convergence of bisection is one: the error is reduced by approximately a factor of 2 with each iteration so that

$$|\epsilon_{n+1}| = \frac{1}{2}\,|\epsilon_n|\,.$$

We now find the order of convergence for Newton's Method and for the Secant Method.

2.4.1. Newton's Method

We start with Newton's Method

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Subtracting both sides from r, we have

$$r - x_{n+1} = r - x_n + \frac{f(x_n)}{f'(x_n)}$$

Or

$$\epsilon_{n+1} = \epsilon_n + \frac{f(x_n)}{f'(x_n)}$$

We use Taylor series to expand the functions $f(x_n)$ and $f'(x_n)$ about the root r, using $f(r) = 0$. We have

$$\begin{aligned} f(x_n) &= f(r) + (x_n - r)\,f'(r) + \frac{1}{2}(x_n - r)^2 f''(r) + \ldots\,, \\ &= -\epsilon_n f'(r) + \frac{1}{2}\epsilon_n^2 f''(r) + \ldots\,; \\ f'(x_n) &= f'(r) + (x_n - r)\,f''(r) + \frac{1}{2}(x_n - r)^2 f'''(r) + \ldots\,, \\ &= f'(r) - \epsilon_n f''(r) + \frac{1}{2}\epsilon_n^2 f'''(r) + \ldots \end{aligned}$$

To make further progress, we will make use of the following standard Taylor series:

$$\frac{1}{1-\epsilon} = 1 + \epsilon + \epsilon^2 + \ldots\,,$$

which converges for $|\epsilon| < 1$. Substituting (2.2) into (2.1), and using (2.3) yields

$$\begin{aligned} \epsilon_{n+1} &= \epsilon_n + \frac{f(x_n)}{f'(x_n)} \\ &= \epsilon_n + \frac{-\epsilon_n f'(r) + \frac{1}{2}\epsilon_n^2 f''(r) + \ldots}{f'(r) - \epsilon_n f''(r) + \frac{1}{2}\epsilon_n^2 f'''(r) + \ldots} \\ &= \epsilon_n + \frac{-\epsilon_n + \frac{1}{2}\epsilon_n^2 \frac{f''(r)}{f'(r)} + \ldots}{1 - \epsilon_n \frac{f''(r)}{f'(r)} + \ldots} \\ &= \epsilon_n + \left(-\epsilon_n + \frac{1}{2}\epsilon_n^2 \frac{f''(r)}{f'(r)} + \ldots\right)\left(1 + \epsilon_n \frac{f''(r)}{f'(r)} + \ldots\right) \\ &= \epsilon_n + \left(-\epsilon_n + \epsilon_n^2\left(\frac{1}{2}\frac{f''(r)}{f'(r)} - \frac{f''(r)}{f'(r)}\right) + \ldots\right) \\ &\quad 1\ f''(r)\ \ 2\ . \end{aligned}$$

$$= -\frac{1}{2}\frac{f''(r)}{f'(r)}\epsilon_n^2 + \ldots$$

Therefore, we have shown that

$$|\epsilon_{n+1}| = k\,|\epsilon_n|^2$$

as $n \to \infty$, with

$$k = \frac{1}{2}\left|\frac{f''(r)}{f'(r)}\right|$$

provided $f'(r) \neq 0$. Newton's method is thus of order 2 at simple roots.

2.4.2. Secant Method

Determining the order of the Secant Method proceeds in a similar fashion. We start with

$$x_{n+1} = x_n - \frac{(x_n - x_{n-1})\,f(x_n)}{f(x_n) - f(x_{n-1})}$$

We subtract both sides from r and make use of

$$\begin{aligned} x_n - x_{n-1} &= (r - x_{n-1}) - (r - x_n) \\ &= \epsilon_{n-1} - \epsilon_n \end{aligned}$$

and the Taylor series

$$\begin{aligned} f(x_n) &= -\epsilon_n f'(r) + \frac{1}{2}\epsilon_n^2 f''(r) + \ldots, \\ f(x_{n-1}) &= -\epsilon_{n-1} f'(r) + \frac{1}{2}\epsilon_{n-1}^2 f''(r) + \ldots, \end{aligned}$$

so that

$$\begin{aligned} f(x_n) - f(x_{n-1}) &= (\epsilon_{n-1} - \epsilon_n)\,f'(r) + \frac{1}{2}\left(\epsilon_n^2 - \epsilon_{n-1}^2\right) f''(r) + \ldots \\ &= (\epsilon_{n-1} - \epsilon_n)\left(f'(r) - \frac{1}{2}(\epsilon_{n-1} + \epsilon_n)\,f''(r) + \ldots\right) \end{aligned}$$

We therefore have

$$\begin{aligned} \epsilon_{n+1} &= \epsilon_n + \frac{-\epsilon_n f'(r) + \frac{1}{2}\epsilon_n^2 f''(r) + \ldots}{f'(r) - \frac{1}{2}(\epsilon_{n-1} + \epsilon_n)\,f''(r) + \ldots} \\ &= \epsilon_n - \epsilon_n \frac{1 - \frac{1}{2}\epsilon_n \frac{f''(r)}{f'(r)} + \ldots}{1 - \frac{1}{2}(\epsilon_{n-1} + \epsilon_n)\frac{f''(r)}{f'(r)} + \ldots} \\ &= \epsilon_n - \epsilon_n \left(1 - \frac{1}{2}\epsilon_n \frac{f''(r)}{f'(r)} + \ldots\right)\left(1 + \frac{1}{2}(\epsilon_{n-1} + \epsilon_n)\frac{f''(r)}{f'(r)} + \ldots\right) \\ &= -\frac{1}{2}\frac{f''(r)}{f'(r)}\epsilon_{n-1}\epsilon_n + \ldots, \end{aligned}$$

or to leading order

$$|\epsilon_{n+1}| = \frac{1}{2}\left|\frac{f''(r)}{f'(r)}\right| |\epsilon_{n-1}|\,|\epsilon_n|$$

The order of convergence is not yet obvious from this equation, and to determine the scaling law we look for a solution of the form

$$|\epsilon_{n+1}| = k\,|\epsilon_n|^p.$$

From this ansatz, we also have

$$|\epsilon_n| = k\,|\epsilon_{n-1}|^p$$

and therefore

$$|\epsilon_{n+1}| = k^{p+1}\,|\epsilon_{n-1}|^{p^2}$$

Substitution into (2.4) results in

$$k^{p+1}\left|\epsilon_{n-1}\right|^{p^2}=\frac{k}{2}\left|\frac{f''(r)}{f'(r)}\right|\left|\epsilon_{n-1}\right|^{p+1}$$

Equating the coefficient and the power of ϵ_{n-1} results in

$$k^p=\frac{1}{2}\left|\frac{f''(r)}{f'(r)}\right|$$

and

$$p^2=p+1$$

The order of convergence of the Secant Method, given by p, therefore is determined to be the positive root of the quadratic equation $p^2-p-1=0$, or

$$p=\frac{1+\sqrt{5}}{2}\approx 1.618$$

which coincidentally is a famous irrational number that is called The Golden Ratio, and goes by the symbol Φ. We see that the Secant Method has an order of convergence lying between the Bisection Method and Newton's Method.

This page titled 2.4: Order of Convergence is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

CHAPTER OVERVIEW

3: System of Equations

Consider the system of n linear equations and n unknowns, given by

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ \vdots & \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned}$$

We can write this system as the matrix equation

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

with

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

3.1: Gaussian Elimination
3.2: LU Decomposition
3.3: Partial Pivoting
3.4: Operation Counts
3.5: System of Nonlinear Equations

This page titled 3: System of Equations is shared under a CC BY 1.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov.

3.1: Gaussian Elimination

The standard numerical algorithm to solve a system of linear equations is called Gaussian Elimination. We can illustrate this algorithm by example.

Consider the system of equations

$$\begin{aligned} -3x_1 + 2x_2 - x_3 &= -1 \\ 6x_1 - 6x_2 + 7x_3 &= -7 \\ 3x_1 - 4x_2 + 4x_3 &= -6 \end{aligned}$$

To perform Gaussian elimination, we form an Augmented Matrix by combining the matrix A with the column vector b :

$$\begin{pmatrix} -3 & 2 & -1 & -1 \\ 6 & -6 & 7 & -7 \\ 3 & -4 & 4 & -6 \end{pmatrix}$$

Row reduction is then performed on this matrix. Allowed operations are (1) multiply any row by a constant, (2) add multiple of one row to another row, (3) interchange the order of any rows. The goal is to convert the original matrix into an upper-triangular matrix

We start with the first row of the matrix and work our way down as follows. First we multiply the first row by 2 and add it to the second row, and add the first row to the third row:

$$\begin{pmatrix} -3 & 2 & -1 & -1 \\ 0 & -2 & 5 & -9 \\ 0 & -2 & 3 & -7 \end{pmatrix}$$

We then go to the second row. We multiply this row by -1 and add it to the third row:

$$\begin{pmatrix} -3 & 2 & -1 & -1 \\ 0 & -2 & 5 & -9 \\ 0 & 0 & -2 & 2 \end{pmatrix}$$

The resulting equations can be determined from the matrix and are given by

$$\begin{aligned} -3x_1 + 2x_2 - x_3 &= -1 \\ -2x_2 + 5x_3 &= -9 \\ -2x_3 &= 2 \end{aligned}$$

These equations can be solved by backward substitution, starting from the last equation and working backwards. We have

$$\begin{aligned} -2x_3 &= 2 \rightarrow x_3 = -1 \\ -2x_2 &= -9 - 5x_3 = -4 \rightarrow x_2 = 2 \\ -3x_1 &= -1 - 2x_2 + x_3 = -6 \rightarrow x_1 = 2 \end{aligned}$$

Therefore,

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 2 \\ -1 \end{pmatrix}$$

This page titled 3.1: Gaussian Elimination is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

3.2: LU Decomposition

The process of Gaussian Elimination also results in the factoring of the matrix A to

$$\mathrm{A} = \mathrm{LU},$$

where L is a lower triangular matrix and U is an upper triangular matrix. Using the same matrix A as in the last section, we show how this factorization is realized. We have

$$\begin{pmatrix} -3 & 2 & -1 \\ 6 & -6 & 7 \\ 3 & -4 & 4 \end{pmatrix} \rightarrow \begin{pmatrix} -3 & 2 & -1 \\ 0 & -2 & 5 \\ 0 & -2 & 3 \end{pmatrix} = \mathrm{M}_1\ \mathrm{A}$$

where

$$\mathrm{M}_1\ \mathrm{A} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} -3 & 2 & -1 \\ 6 & -6 & 7 \\ 3 & -4 & 4 \end{pmatrix} = \begin{pmatrix} -3 & 2 & -1 \\ 0 & -2 & 5 \\ 0 & -2 & 3 \end{pmatrix}$$

Note that the matrix M_1 performs row elimination on the first column. Two times the first row is added to the second row and one times the first row is added to the third row. The entries of the column of M_1 come from $2 = -(6/-3)$ and $1 = -(3/-3)$ as required for row elimination. The number -3 is called the pivot.

The next step is

$$\begin{pmatrix} -3 & 2 & -1 \\ 0 & -2 & 5 \\ 0 & -2 & 3 \end{pmatrix} \rightarrow \begin{pmatrix} -3 & 2 & -1 \\ 0 & -2 & 5 \\ 0 & 0 & -2 \end{pmatrix} = \mathrm{M}_2\left(\mathrm{M}_1\ \mathrm{A}\right)$$

where

$$\mathrm{M}_2\left(\mathrm{M}_1\ \mathrm{A}\right) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} -3 & 2 & -1 \\ 0 & -2 & 5 \\ 0 & -2 & 3 \end{pmatrix} = \begin{pmatrix} -3 & 2 & -1 \\ 0 & -2 & 5 \\ 0 & 0 & -2 \end{pmatrix}$$

Here, M_2 multiplies the second row by $-1 = -(-2/-2)$ and adds it to the third row. The pivot is -2.

We now have

$$\mathrm{M}_2\mathrm{M}_1\ \mathrm{A} = \mathrm{U}$$

or

$$\mathrm{A} = \mathrm{M}_1^{-1}\mathrm{M}_2^{-1}\mathrm{U}$$

The inverse matrices are easy to find. The matrix M_1 multiples the first row by 2 and adds it to the second row, and multiplies the first row by 1 and adds it to the third row. To invert these operations, we need to multiply the first row by -2 and add it to the second row, and multiply the first row by -1 and add it to the third row. To check, with

$$\mathrm{M}_1\mathrm{M}_1^{-1} = \mathrm{I}$$

we have

$$\begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Similarly,

$$\mathrm{M}_2^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

Therefore,

$$\mathrm{L} = \mathrm{M}_1^{-1}\mathrm{M}_2^{-1}$$

is given by

$$L = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -1 & 1 & 1 \end{pmatrix}$$

which is lower triangular. The off-diagonal elements of M_1^{-1} and M_2^{-1} are simply combined to form L. Our LU decomposition is therefore

$$\begin{pmatrix} -3 & 2 & -1 \\ 6 & -6 & 7 \\ 3 & -4 & 4 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -1 & 1 & 1 \end{pmatrix} \begin{pmatrix} -3 & 2 & -1 \\ 0 & -2 & 5 \\ 0 & 0 & -2 \end{pmatrix}.$$

Another nice feature of the LU decomposition is that it can be done by overwriting A, therefore saving memory if the matrix A is very large.

The LU decomposition is useful when one needs to solve $A\mathbf{x} = \mathbf{b}$ for $\mathbf{x}$ when A is fixed and there are many different b's. First one determines L and U using Gaussian elimination. Then one writes

$$(\mathrm{LU})\mathbf{x} = \mathrm{L}(\mathrm{U}\mathbf{x}) = \mathbf{b}.$$

We let

$$\mathbf{y} = \mathbf{U}\mathbf{x}$$

and first solve

$$\mathrm{L}\mathrm{y} = \mathbf{b}$$

for y by forward substitution. We then solve

$$\mathrm{U}\mathrm{x} = \mathbf{y}$$

for x by backward substitution. When we count operations, we will see that solving $(\mathrm{LU})\mathrm{x} = \mathbf{b}$ is significantly faster once L and U are in hand than solving $\mathrm{A}\mathrm{x} = \mathbf{b}$ directly by Gaussian elimination.

We now illustrate the solution of $LU\mathbf{x} = \mathbf{b}$ using our previous example, where

$$\mathrm{L} = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -1 & 1 & 1 \end{pmatrix}, \quad \mathrm{U} = \begin{pmatrix} -3 & 2 & -1 \\ 0 & -2 & 5 \\ 0 & 0 & -2 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} -1 \\ -7 \\ -6 \end{pmatrix}$$

With $\mathbf{y} = \mathbf{U}\mathrm{x}$, we first solve $\mathbf{L}\mathbf{y} = \mathrm{b}$, that is

$$\begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -1 & 1 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} -1 \\ -7 \\ -6 \end{pmatrix}$$

Using forward substitution

$$\begin{aligned} y_1 &= -1 \\ y_2 &= -7 + 2y_1 = -9 \\ y_3 &= -6 + y_1 - y_2 = 2 \end{aligned}$$

We now solve $Ux = y$, that is

$$\begin{pmatrix} -3 & 2 & -1 \\ 0 & -2 & 5 \\ 0 & 0 & -2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} -1 \\ -9 \\ 2 \end{pmatrix}$$

Using backward substitution,

$$\begin{aligned} -2x_3 &= 2 \rightarrow x_3 = -1 \\ -2x_2 &= -9 - 5x_3 = -4 \rightarrow x_2 = 2 \end{aligned}$$

$$-3x_1 = -1 - 2x_2 + x_3 = -6 \rightarrow x_1 = 2$$

and we have once again determined

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 2 \\ -1 \end{pmatrix}$$

This page titled 3.2: LU Decomposition is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

3.3: Partial Pivoting

When performing Gaussian elimination, the diagonal element that one uses during the elimination procedure is called the **pivot**. To obtain the correct multiple, one uses the pivot as the divisor to the elements below the pivot. Gaussian elimination in this form will fail if the pivot is zero. In this situation, a row interchange must be performed.

Even if the pivot is not identically zero, a small value can result in big roundoff errors. For very large matrices, one can easily lose all accuracy in the solution. To avoid these round-off errors arising from small pivots, row interchanges are made, and this technique is called **partial pivoting** (partial pivoting is in contrast to complete pivoting, where both rows and columns are interchanged). We will illustrate by example the LU decomposition using partial pivoting. Consider

$$A = \begin{pmatrix} -2 & 2 & -1 \\ 6 & -6 & 7 \\ 3 & -8 & 4 \end{pmatrix}$$

We interchange rows to place the largest element (in absolute value) in the pivot, or a_{11}, position. That is,

$$\mathrm{A} \rightarrow \begin{pmatrix} 6 & -6 & 7 \\ -2 & 2 & -1 \\ 3 & -8 & 4 \end{pmatrix} = \mathrm{P}_{12}\ \mathrm{A}$$

where

$$P_{12} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

is a permutation matrix that when multiplied on the left interchanges the first and second rows of a matrix. Note that $P_{12}^{-1} = P_{12}$. The elimination step is then

$$\mathrm{P}_{12}\ \mathrm{A} \rightarrow \begin{pmatrix} 6 & -6 & 7 \\ 0 & 0 & 4/3 \\ 0 & -5 & 1/2 \end{pmatrix} = \mathrm{M}_1\mathrm{P}_{12}\ \mathrm{A}$$

where

$$M_1 = \begin{pmatrix} 1 & 0 & 0 \\ 1/3 & 1 & 0 \\ -1/2 & 0 & 1 \end{pmatrix}$$

The final step requires one more row interchange:

$$\mathrm{M}_1\mathrm{P}_{12}\ \mathrm{A} \rightarrow \begin{pmatrix} 6 & -6 & 7 \\ 0 & -5 & 1/2 \\ 0 & 0 & 4/3 \end{pmatrix} = \mathrm{P}_{23}\mathrm{M}_1\mathrm{P}_{12}\ \mathrm{A} = \mathrm{U}$$

Since the permutation matrices given by P are their own inverses, we can write our result as

$$\left(P_{23}M_1P_{23}\right) P_{23}P_{12}A = U$$

Multiplication of M on the left by P interchanges rows while multiplication on the right by P interchanges columns. That is,

$$P_{23}\begin{pmatrix} 1 & 0 & 0 \\ 1/3 & 1 & 0 \\ -1/2 & 0 & 1 \end{pmatrix} P_{23} = \begin{pmatrix} 1 & 0 & 0 \\ -1/2 & 0 & 1 \\ 1/3 & 1 & 0 \end{pmatrix} P_{23} = \begin{pmatrix} 1 & 0 & 0 \\ -1/2 & 1 & 0 \\ 1/3 & 0 & 1 \end{pmatrix}$$

The net result on M_1 is an interchange of the nondiagonal elements $1/3$ and $-1/2$.

We can then multiply by the inverse of $\left(\mathrm{P}_{23}\mathrm{M}_1\mathrm{P}_{23}\right)$ to obtain

$$P_{23}P_{12}A = \left(P_{23}M_1P_{23}\right)^{-1} U$$

which we write as

$$\mathrm{PA} = \mathrm{LU}$$

Instead of L, MATLAB will write this as

$$\mathrm{A} = \left(\mathrm{P}^{-1}\ \mathrm{L}\right) \mathrm{U}$$

For convenience, we will just denote $\left(\mathrm{P}^{-1}\ \mathrm{L}\right)$ by L, but by L here we mean a permutated lower triangular matrix.

For example, in MATLAB, to solve $A\mathbf{x} = \mathbf{b}$ for x using Gaussian elimination, one types

$$\mathrm{x} = \mathrm{A} \backslash \mathrm{b}$$

where \solves for x using the most efficient algorithm available, depending on the form of A. If A is a general $n \times n$ matrix, then first the LU decomposition of A is found using partial pivoting, and then x is determined from permuted forward and backward substitution. If A is upper or lower triangular, then forward or backward substitution (or their permuted version) is used directly.

If there are many different right-hand-sides, one would first directly find the LU decomposition of A using a function call, and then solve using . That is, one would iterate for different b' s the following expressions:

$$\begin{aligned} [\mathrm{LU}] &= lu(\ \mathrm{A}) \\ \mathrm{y} &= \mathrm{L} \backslash \mathrm{b} \\ \mathrm{x} &= \mathrm{U} \backslash y \end{aligned}$$

where the second and third lines can be shortened to

$$\mathrm{x} = \mathrm{U} \backslash (\mathrm{L} \backslash \mathrm{b})$$

where the parenthesis are required. In lecture, I will demonstrate these solutions in MATLAB using the matrix A $= [-2, 2, -1; 6, -6, 7; 3, -8, 4]$; which is the example in the notes.

This page titled 3.3: Partial Pivoting is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

3.4: Operation Counts

To estimate how much computational time is required for an algorithm, one can count the number of operations required (multiplications, divisions, additions and subtractions). Usually, what is of interest is how the algorithm scales with the size of the problem. For example, suppose one wants to multiply two full $n \times n$ matrices. The calculation of each element requires n multiplications and $n-1$ additions, or say $2n-1$ operations. There are n^2 elements to compute so that the total operation count is $n^2(2n-1)$. If n is large, we might want to know what will happen to the computational time if n is doubled. What matters most is the fastest-growing, leading-order term in the operation count. In this matrix multiplication example, the operation count is $n^2(2n-1) = 2n^3 - n^2$, and the leading-order term is $2n^3$. The factor of 2 is unimportant for the scaling, and we say that the algorithm scales like $\mathrm{O}\left(n^3\right)$, which is read as "big Oh of n cubed." When using the big-Oh notation, we will drop both lower-order terms and constant multipliers.

The big-Oh notation tells us how the computational time of an algorithm scales. For example, suppose that the multiplication of two large $n \times n$ matrices took a computational time of T_n seconds. With the known operation count going like $\mathrm{O}\left(n^3\right)$, we can write

$$T_n = kn^3$$

for some unknown constant k. To determine how much longer the multiplication of two $2n \times 2n$ matrices will take, we write

$$\begin{aligned} T_{2n} &= k(2n)^3 \\ &= 8kn^3 \\ &= 8T_n \end{aligned}$$

so that doubling the size of the matrix is expected to increase the computational time by a factor of $2^3 = 8$.

Running MATLAB on my computer, the multiplication of two 2048×2048 matrices took about $0.75\mathrm{sec}$. The multiplication of two 4096×4096 matrices took about $6\mathrm{sec}$, which is 8 times longer. Timing of code in MATLAB can be found using the built-in stopwatch functions tic and toc.

What is the operation count and therefore the scaling of Gaussian elimination? Consider an elimination step with the pivot in the i th row and i th column. There are both $n-i$ rows below the pivot and $n-i$ columns to the right of the pivot. To perform elimination of one row, each matrix element to the right of the pivot must be multiplied by a factor and added to the row underneath. This must be done for all the rows. There are therefore $(n-i)(n-i)$ multiplication-additions to be done for this pivot. Since we are interested in only the scaling of the algorithm, I will just count a multiplication-addition as one operation.

To find the total operation count, we need to perform elimination using $n-1$ pivots, so that

$$\begin{aligned} \text{op. counts } &= \sum_{i=1}^{n-1} (n-i)^2 \\ &= (n-1)^2 + (n-2)^2 + \ldots (1)^2 \\ &= \sum_{i=1}^{n-1} i^2 \end{aligned}$$

Three summation formulas will come in handy. They are

$$\begin{aligned} \sum_{i=1}^{n} 1 &= n \\ \sum_{i=1}^{n} i &= \frac{1}{2} n(n+1) \\ \sum_{i=1}^{n} i^2 &= \frac{1}{6} n(2n+1)(n+1) \end{aligned}$$

which can be proved by mathematical induction, or derived by some tricks.

The operation count for Gaussian elimination is therefore

$$\begin{aligned} \text{op. counts } &= \sum_{i=1}^{n-1} i^2 \\ &= \frac{1}{6}(n-1)(2n-1)(n) \end{aligned}$$

The leading-order term is therefore $n^3/3$, and we say that Gaussian elimination scales like $\mathrm{O}\left(n^3\right)$. Since LU decomposition with partial pivoting is essentially Gaussian elimination, that algorithm also scales like $\mathrm{O}\left(n^3\right)$. However, once the LU decomposition of a matrix A is known, the solution of $\mathbf{Ax} = \mathbf{b}$ can proceed by a forward and backward substitution. How does a backward substitution, say, scale? For backward substitution, the matrix equation to be solved is of the form

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n-1} & a_{1,n} \\ 0 & a_{2,2} & \cdots & a_{2,n-1} & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & a_{n-1,n-1} & a_{n-1,n} \\ 0 & 0 & \cdots & 0 & a_{n,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n-1} \\ b_n \end{pmatrix}$$

The solution for x_i is found after solving for x_j with $j > i$. The explicit solution for x_i is given by

$$x_i = \frac{1}{a_{i,i}} \left(b_i - \sum_{j=i+1}^{n} a_{i,j} x_j \right)$$

The solution for x_i requires $n - i + 1$ multiplication-additions, and since this must be done for n such i' s, we have

$$\begin{aligned} \text{op. counts } &= \sum_{i=1}^{n} n - i + 1 \\ &= n + (n-1) + \cdots + 1 \\ &= \sum_{i=1}^{n} i \\ &= \frac{1}{2} n(n+1) \end{aligned}$$

The leading-order term is $n^2/2$ and the scaling of backward substitution is $\mathrm{O}\left(n^2\right)$. After the LU decomposition of a matrix A is found, only a single forward and backward substitution is required to solve $A\mathbf{x} = \mathbf{b}$, and the scaling of the algorithm to solve this matrix equation is therefore still $\mathrm{O}\left(n^2\right)$. For large n, one should expect that solving $\mathbf{Ax} = \mathbf{b}$ by a forward and backward substitution should be substantially faster than a direct solution using Gaussian elimination.

This page titled 3.4: Operation Counts is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

3.5: System of Nonlinear Equations

A system of nonlinear equations can be solved using a version of Newton's Method. We illustrate this method for a system of two equations and two unknowns. Suppose that we want to solve

$$f(x,y)=0, \quad g(x,y)=0$$

for the unknowns x and y. We want to construct two simultaneous sequences $x_0, x_1, x_2, ...$ and $y_0, y_1, y_2, ...$ that converge to the roots. To construct these sequences, we Taylor series expand $f(x_{n+1}, y_{n+1})$ and $g(x_{n+1}, y_{n+1})$ about the point (x_n, y_n). Using the partial derivatives $f_x = \partial f/\partial x, f_y = \partial f/\partial y$, etc., the twodimensional Taylor series expansions, displaying only the linear terms, are given by

$$f(x_{n+1}, y_{n+1}) = f(x_n, y_n) + (x_{n+1} - x_n) f_x(x_n, y_n) + (y_{n+1} - y_n) f_y(x_n, y_n) + ...$$

$$g(x_{n+1}, y_{n+1}) = g(x_n, y_n) + (x_{n+1} - x_n) g_x(x_n, y_n) + (y_{n+1} - y_n) g_y(x_n, y_n) + ...$$

To obtain Newton's method, we take $f(x_{n+1}, y_{n+1}) = 0, g(x_{n+1}, y_{n+1}) = 0$ and drop higher-order terms above linear. Although one can then find a system of linear equations for x_{n+1} and y_{n+1}, it is more convenient to define the variables

$$\Delta x_n = x_{n+1} - x_n, \quad \Delta y_n = y_{n+1} - y_n.$$

The iteration equations will then be given by

$$x_{n+1} = x_n + \Delta x_n, \quad y_{n+1} = y_n + \Delta y_n$$

and the linear equations to be solved for Δx_n and Δy_n are given by

$$\begin{pmatrix} f_x & f_y \\ g_x & g_y \end{pmatrix} \begin{pmatrix} \Delta x_n \\ \Delta y_n \end{pmatrix} = \begin{pmatrix} -f \\ -g \end{pmatrix}$$

where f, g, f_x, f_y, g_x, and g_y are all evaluated at the point (x_n, y_n). The twodimensional case is easily generalized to n dimensions. The matrix of partial derivatives is called the Jacobian Matrix.

We illustrate Newton's Method by finding the steady state solution of the Lorenz equations, given by

$$\begin{aligned} \sigma(y-x) &= 0 \\ rx - y - xz &= 0 \\ xy - bz &= 0 \end{aligned}$$

where x, y, and z are the unknown variables and σ, r, and b are the known parameters. Here, we have a three-dimensional homogeneous system $f = 0, g = 0$, and $h = 0$, with

$$\begin{aligned} f(x,y,z) &= \sigma(y-x) \\ g(x,y,z) &= rx - y - xz \\ h(x,y,z) &= xy - bz. \end{aligned}$$

The partial derivatives can be computed to be

$$\begin{aligned} f_x &= -\sigma, & f_y &= \sigma, & f_z &= 0, \\ g_x &= r - z, & g_y &= -1, & g_z &= -x, \\ h_x &= y, & h_y &= x, & h_z &= -b. \end{aligned}$$

The iteration equation is therefore

$$\begin{pmatrix} -\sigma & \sigma & 0 \\ r - z_n & -1 & -x_n \\ y_n & x_n & -b \end{pmatrix} \begin{pmatrix} \Delta x_n \\ \Delta y_n \\ \Delta z_n \end{pmatrix} = - \begin{pmatrix} \sigma(y_n - x_n) \\ rx_n - y_n - x_n z_n \\ x_n y_n - b z_n \end{pmatrix}$$

with

$$x_{n+1} = x_n + \Delta x_n$$

$$y_{n+1} = y_n + \Delta y_n$$
$$z_{n+1} = z_n + \Delta z_n$$

The MATLAB program that solves this system is contained in newton_system.m.

This page titled 3.5: System of Nonlinear Equations is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

CHAPTER OVERVIEW

4: Least-Squares Approximation

The method of least-squares is commonly used to fit a parameterized curve to experimental data. In general, the fitting curve is not expected to pass through the data points, making this problem substantially different from the one of interpolation. We consider here only the simplest case of the same experimental error for all the data points. Let the data to be fitted be given by (x_i, y_i), with $i = 1$ to n.

4.1: Fitting a Straight Line

4.2: Fitting to a Linear Combination of Functions

This page titled 4: Least-Squares Approximation is shared under a CC BY 1.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov.

4.1: Fitting a Straight Line

Suppose the fitting curve is a line. We write for the fitting curve

$$y(x) = \alpha x + \beta.$$

The distance r_i from the data point (x_i, y_i) and the fitting curve is given by

$$\begin{aligned} r_i &= y_i - y\left(x_i\right) \\ &= y_i - \left(\alpha x_i + \beta\right). \end{aligned}$$

A least-squares fit minimizes the sum of the squares of the r_i 's. This minimum can be shown to result in the most probable values of α and β.

We define

$$\begin{aligned} \rho &= \sum_{i=1}^{n} r_i^2 \\ &= \sum_{i=1}^{n} \left(y_i - \left(\alpha x_i + \beta\right)\right)^2 \end{aligned}$$

To minimize ρ with respect to α and β, we solve

$$\frac{\partial \rho}{\partial \alpha} = 0, \quad \frac{\partial \rho}{\partial \beta} = 0$$

Taking the partial derivatives, we have

$$\begin{aligned} \frac{\partial \rho}{\partial \alpha} &= \sum_{i=1}^{n} 2\left(-x_i\right)\left(y_i - \left(\alpha x_i + \beta\right)\right) = 0 \\ \frac{\partial \rho}{\partial \beta} &= \sum_{i=1}^{n} 2(-1)\left(y_i - \left(\alpha x_i + \beta\right)\right) = 0 \end{aligned}$$

These equations form a system of two linear equations in the two unknowns α and β, which is evident when rewritten in the form

$$\begin{aligned} \alpha \sum_{i=1}^{n} x_i^2 + \beta \sum_{i=1}^{n} x_i &= \sum_{i=1}^{n} x_i y_i \\ \alpha \sum_{i=1}^{n} x_i + \beta n &= \sum_{i=1}^{n} y_i \end{aligned}$$

These equations can be solved either analytically, or numerically in MATLAB, where the matrix form is

$$\begin{pmatrix} \sum_{i=1}^{n} & x_i^2 & \sum_{i=1}^{n} \\ \sum_{i=1}^{n} & x_i & n \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{n} x_i y_i \\ \sum_{i=1}^{n} y_i \end{pmatrix}.$$

A proper statistical treatment of this problem should also consider an estimate of the errors in α and β as well as an estimate of the goodness-of-fit of the data to the model. We leave these further considerations to a statistics class.

This page titled 4.1: Fitting a Straight Line is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

4.2: Fitting to a Linear Combination of Functions

Consider the general fitting function

$$y(x) = \sum_{j=1}^{m} c_j f_j(x)$$

where we assume m functions $f_j(x)$. For example, if we want to fit a cubic polynomial to the data, then we would have $m = 4$ and take $f_1 = 1, f_2 = x, f_3 = x^2$ and $f_4 = x^3$. Typically, the number of functions f_j is less than the number of data points; that is, $m < n$, so that a direct attempt to solve for the c_j' s such that the fitting function exactly passes through the n data points would result in n equations and m unknowns. This would be an over-determined linear system that in general has no solution.

We now define the vectors

$$\mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}, \quad \mathbf{c} = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{pmatrix}$$

and the n-by-m matrix

$$\mathrm{A} = \begin{pmatrix} f_1(x_1) & f_2(x_1) & \cdots & f_m(x_1) \\ f_1(x_2) & f_2(x_2) & \cdots & f_m(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_n) & f_2(x_n) & \cdots & f_m(x_n) \end{pmatrix}$$

With $m < n$, the equation $A\mathbf{c} = \mathbf{y}$ is over-determined. We let

$$\mathbf{r} = \mathbf{y} - \mathrm{A}\mathbf{c}$$

be the residual vector, and let

$$\rho = \sum_{i=1}^{n} r_i^2$$

The method of least squares minimizes ρ with respect to the components of c. Now, using T to signify the transpose of a matrix, we have

$$\begin{aligned} \rho &= \mathbf{r}^T\mathbf{r} \\ &= (\mathbf{y} - \mathrm{A}\mathbf{c})^T(\mathbf{y} - \mathrm{A}\mathbf{c}) \\ &= \mathbf{y}^T\mathbf{y} - \mathbf{c}^T\,\mathrm{A}^T\mathbf{y} - \mathbf{y}^T\,\mathrm{A}\mathbf{c} + \mathbf{c}^T\,\mathrm{A}^T\,\mathrm{A}\mathbf{c}. \end{aligned}$$

Since ρ is a scalar, each term in the above expression must be a scalar, and since the transpose of a scalar is equal to the scalar, we have

$$\mathbf{c}^T\,\mathrm{A}^T\mathbf{y} = \left(\mathbf{c}^T\,\mathrm{A}^T\mathbf{y}\right)^T = \mathbf{y}^T\,\mathrm{A}\mathbf{c}.$$

Therefore,

$$\rho = \mathbf{y}^T\mathbf{y} - 2\mathbf{y}^T\,\mathrm{A}\mathbf{c} + \mathbf{c}^T\,\mathrm{A}^T\,\mathrm{A}\mathbf{c}.$$

To find the minimum of ρ, we will need to solve $\partial\rho/\partial c_j = 0$ for $j = 1, \ldots, m$. To take the derivative of ρ, we switch to a tensor notation, using the Einstein summation convention, where repeated indices are summed over their allowable range. We can write

$$\rho = y_i y_i - 2y_i\,\mathrm{A}_{ik}c_k + c_i\,\mathrm{A}_{ik}^T\,\mathrm{A}_{kl}c_l$$

Taking the partial derivative, we have

$$\frac{\partial\rho}{\partial c_j} = -2y_i\,\mathrm{A}_{ik}\frac{\partial c_k}{\partial c_j} + \frac{\partial c_i}{\partial c_j}\,\mathrm{A}_{ik}^T\,\mathrm{A}_{kl}c_l + c_i\,\mathrm{A}_{ik}^T\,\mathrm{A}_{kl}\frac{\partial c_l}{\partial c_j}$$

Now,

$$\frac{\partial c_i}{\partial c_j} = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{otherwise} \end{cases}$$

Therefore,

$$\frac{\partial \rho}{\partial c_j} = -2y_i \, \mathrm{A}_{ij} + \mathrm{A}^T_{jk} \, \mathrm{A}_{kl} c_l + c_i \, \mathrm{A}^T_{ik} \, \mathrm{A}_{kj}$$

Now,

$$\begin{aligned} c_i \, \mathrm{A}^T_{ik} \, \mathrm{A}_{kj} &= c_i \, \mathrm{A}_{ki} \, \mathrm{A}_{kj} \\ &= \mathrm{A}_{kj} \, \mathrm{A}_{ki} c_i \\ &= \mathrm{A}^T_{jk} \, \mathrm{A}_{ki} c_i \\ &= \mathrm{A}^T_{jk} \, \mathrm{A}_{kl} c_l \end{aligned}$$

Therefore,

$$\frac{\partial \rho}{\partial c_j} = -2y_i \, \mathrm{A}_{ij} + 2 \, \mathrm{A}^T_{jk} \, \mathrm{A}_{kl} c_l$$

With the partials set equal to zero, we have

$$\mathrm{A}^T_{jk} \, \mathrm{A}_{kl} c_l = y_i \, \mathrm{A}_{ij}$$

or

$$\mathrm{A}^T_{jk} \, \mathrm{A}_{kl} c_l = \mathrm{A}^T_{ji} y_i$$

In vector notation, we have

$$\mathrm{A}^T \, \mathrm{A}\mathbf{c} = \mathrm{A}^T \mathbf{y}.$$

Equation (4.2) is the so-called normal equation, and can be solved for c by Gaussian elimination using the MATLAB backslash operator. After constructing the matrix A given by (4.1), and the vector y from the data, one can code in MATLAB

$$c = (A'A) \setminus (A'y)$$

But in fact the MATLAB back slash operator will automatically solve the normal equations when the matrix A is not square, so that the MATLAB code

$$c = A \setminus y$$

yields the same result.

This page titled 4.2: Fitting to a Linear Combination of Functions is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

CHAPTER OVERVIEW

5: Interpolation

Consider the following problem: Given the values of a known function $y = f(x)$ at a sequence of ordered points $x_0, x_1, \ldots, x_n$, find $f(x)$ for arbitrary x. When $x_0 \leq x \leq x_n$, the problem is called interpolation. When $x < x_0$ or $x > x_n$ the problem is called extrapolation.

With $y_i = f(x_i)$, the problem of interpolation is basically one of drawing a smooth curve through the known points $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$. This is not the same problem as drawing a smooth curve that approximates a set of data points that have experimental error. This latter problem is called least-squares approximation.

Here, we will consider three interpolation algorithms: (1) polynomial interpolation; (2) piecewise linear interpolation, and; (3) cubic spline interpolation.

5.1: Polynomial Interpolation
5.2: Piecewise Linear Interpolation
5.3: Cubic Spline Interpolation
5.4: Multidimensional Interpolation

This page titled 5: Interpolation is shared under a CC BY 1.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov.

5.1: Polynomial Interpolation

The $n+1$ points $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$ can be interpolated by a unique polynomial of degree n. When $n = 1$, the polynomial is a linear function; when $n = 2$, the polynomial is a quadratic function. There are three standard algorithms that can be used to construct this unique interpolating polynomial, and we will present all three here, not so much because they are all useful, but because it is interesting to learn how these three algorithms are constructed.

When discussing each algorithm, we define $P_n(x)$ to be the unique nth degree polynomial that passes through the given $n+1$ data points.

5.1.1. Vandermonde polynomial

This Vandermonde polynomial is the most straightforward construction of the interpolating polynomial $P_n(x)$. We simply write

$$P_n(x) = c_0 x^n + c_1 x^{n-1} + \cdots + c_n.$$

Then we can immediately form $n+1$ linear equations for the $n+1$ unknown coefficients $c_0, c_1, \ldots, c_n$ using the $n+1$ known points:

$$\begin{aligned} y_0 &= c_0 x_0^n + c_1 x_0^{n-1} + \cdots + c_{n-1} x_0 + c_n \\ y_2 &= c_0 x_1^n + c_1 x_1^{n-1} + \cdots + c_{n-1} x_1 + c_n \\ &\vdots \\ y_n &= c_0 x_n^n + c_1 x_n^{n-1} + \cdots + c_{n-1} x_n + c_n \end{aligned}$$

The system of equations in matrix form is

$$\begin{pmatrix} x_0^n & x_0^{n-1} & \cdots & x_0 & 1 \\ x_1^n & x_1^{n-1} & \cdots & x_1 & 1 \\ \vdots & \vdots & \ddots & \vdots & \\ x_n^n & x_n^{n-1} & \cdots & x_n & 1 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix}$$

The matrix is called the **Vandermonde matrix**, and can be constructed using the MATLAB function vander.m. The system of linear equations can be solved in MATLAB using the \operator, and the MATLAB function polyval.m can used to interpolate using the c coefficients. I will illustrate this in class and place the code on the website.

5.1.2. Lagrange polynomial

The Lagrange polynomial is the most clever construction of the interpolating polynomial $P_n(x)$, and leads directly to an analytical formula. The Lagrange polynomial is the sum of $n+1$ terms and each term is itself a polynomial of degree n. The full polynomial is therefore of degree n. Counting from 0 , the i th term of the Lagrange polynomial is constructed by requiring it to be zero at x_j with $j \neq i$, and equal to y_i when $j = i$. The polynomial can be written as

$$\begin{aligned} P_n(x) = \frac{(x-x_1)(x-x_2)\cdots(x-x_n)\,y_0}{(x_0-x_1)(x_0-x_2)\cdots(x_0-x_n)} + \frac{(x-x_0)(x-x_2)\cdots(x-x_n)\,y_1}{(x_1-x_0)(x_1-x_2)\cdots(x_1-x_n)} \\ + \cdots + \frac{(x-x_0)(x-x_1)\cdots(x-x_{n-1})\,y_n}{(x_n-x_0)(x_n-x_1)\cdots(x_n-x_{n-1})}. \end{aligned}$$

It can be clearly seen that the first term is equal to zero when $x = x_1, x_2, \ldots, x_n$ and equal to y_0 when $x = x_0$; the second term is equal to zero when $x = x_0, x_2, \ldots x_n$ and equal to y_1 when $x = x_1$; and the last term is equal to zero when $x = x_0, x_1, \ldots x_{n-1}$ and equal to y_n when $x = x_n$. The uniqueness of the interpolating polynomial implies that the Lagrange polynomial must be the interpolating polynomial.

5.1.3. Newton polynomial

The Newton polynomial is somewhat more clever than the Vandermonde polynomial because it results in a system of linear equations that is lower triangular, and therefore can be solved by forward substitution. The interpolating polynomial is written in the form

$$P_n(x) = c_0 + c_1(x-x_0) + c_2(x-x_0)(x-x_1) + \cdots + c_n(x-x_0)\cdots(x-x_{n-1})$$

which is clearly a polynomial of degree n. The $n+1$ unknown coefficients given by the c' s can be found by substituting the points (x_i, y_i) for $i = 0, \ldots, n$:

$$\begin{aligned}
y_0 &= c_0 \\
y_1 &= c_0 + c_1 (x_1 - x_0) \\
y_2 &= c_0 + c_1 (x_2 - x_0) + c_2 (x_2 - x_0)(x_2 - x_1) \\
&\vdots \\
y_n &= c_0 + c_1 (x_n - x_0) + c_2 (x_n - x_0)(x_n - x_1) + \cdots + c_n (x_n - x_0) \cdots (x_n - x_{n-1})
\end{aligned}$$

This system of linear equations is lower triangular as can be seen from the matrix form

$$\begin{pmatrix}
1 & 0 & 0 & \cdots & 0 \\
1 & (x_1 - x_0) & 0 & \cdots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & (x_n - x_0) & (x_n - x_0)(x_n - x_1) & \cdots & (x_n - x_0) \cdots (x_n - x_{n-1})
\end{pmatrix}
\begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{pmatrix}$$

and so theoretically can be solved faster than the Vandermonde polynomial. In practice, however, there is little difference because polynomial interpolation is only useful when the number of points to be interpolated is small.

This page titled 5.1: Polynomial Interpolation is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

5.2: Piecewise Linear Interpolation

Instead of constructing a single global polynomial that goes through all the points, one can construct local polynomials that are then connected together. In the the section following this one, we will discuss how this may be done using cubic polynomials. Here, we discuss the simpler case of linear polynomials. This is the default interpolation typically used when plotting data.

Suppose the interpolating function is $y = g(x)$, and as previously, there are $n + 1$ points to interpolate. We construct the function $g(x)$ out of n local linear polynomials. We write

$$g(x) = g_i(x), \quad \text{for } x_i \leq x \leq x_{i+1}$$

where

$$g_i(x) = a_i (x - x_i) + b_i$$

and $i = 0, 1, \ldots, n - 1$.

We now require $y = g_i(x)$ to pass through the endpoints (x_i, y_i) and (x_{i+1}, y_{i+1}). We have

$$\begin{aligned} y_i &= b_i \\ y_{i+1} &= a_i (x_{i+1} - x_i) + b_i. \end{aligned}$$

Therefore, the coefficients of $g_i(x)$ are determined to be

$$a_i = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}, \quad b_i = y_i$$

Although piecewise linear interpolation is widely used, particularly in plotting routines, it suffers from a discontinuity in the derivative at each point. This results in a function which may not look smooth if the points are too widely spaced. We next consider a more challenging algorithm that uses cubic polynomials.

This page titled 5.2: Piecewise Linear Interpolation is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

5.3: Cubic Spline Interpolation

The $n+1$ points to be interpolated are again

$$(x_0, y_0), (x_1, y_1), \ldots (x_n, y_n)$$

Here, we use n piecewise cubic polynomials for interpolation,

$$g_i(x) = a_i (x - x_i)^3 + b_i (x - x_i)^2 + c_i (x - x_i) + d_i, \quad i = 0, 1, \ldots, n-1,$$

with the global interpolation function written as

$$g(x) = g_i(x), \qquad \text{for } x_i \le x \le x_{i+1}.$$

To achieve a smooth interpolation we impose that $g(x)$ and its first and second derivatives are continuous. The requirement that $g(x)$ is continuous (and goes through all $n+1$ points) results in the two constraints

$$\begin{aligned} g_i(x_i) &= y_i, \quad i = 0 \text{ to } n-1 \\ g_i(x_{i+1}) &= y_{i+1}, \quad i = 0 \text{ to } n-1 \end{aligned}$$

The requirement that $g'(x)$ is continuous results in

$$g_i'(x_{i+1}) = g_{i+1}'(x_{i+1}), \quad i = 0 \text{ to } n-2$$

And the requirement that $g''(x)$ is continuous results in

$$g_i''(x_{i+1}) = g_{i+1}''(x_{i+1}), \quad i = 0 \text{ to } n-2$$

There are n cubic polynomials $g_i(x)$ and each cubic polynomial has four free coefficients; there are therefore a total of $4n$ unknown coefficients. The number of constraining equations from (5.1)-(5.4) is $2n + 2(n-1) = 4n - 2$. With $4n-2$ constraints and $4n$ unknowns, two more conditions are required for a unique solution. These are usually chosen to be extra conditions on the first $g_0(x)$ and last $g_{n-1}(x)$ polynomials. We will discuss these extra conditions later.

We now proceed to determine equations for the unknown coefficients of the cubic polynomials. The polynomials and their first two derivatives are given by

$$\begin{aligned} g_i(x) &= a_i (x - x_i)^3 + b_i (x - x_i)^2 + c_i (x - x_i) + d_i \\ g_i'(x) &= 3a_i (x - x_i)^2 + 2b_i (x - x_i) + c_i \\ g_i''(x) &= 6a_i (x - x_i) + 2b_i \end{aligned}$$

We will consider the four conditions (5.1)-(5.4) in turn. From (5.1) and (5.5), we have

$$d_i = y_i, \quad i = 0 \text{ to } n-1,$$

which directly solves for all of the d-coefficients.

To satisfy (5.2), we first define

$$h_i = x_{i+1} - x_i$$

and

$$f_i = y_{i+1} - y_i$$

Now, from (5.2) and (5.5), using (5.8), we obtain the n equations

$$a_i h_i^3 + b_i h_i^2 + c_i h_i = f_i, \quad i = 0 \text{ to } n-1$$

From (5.3) and (5.6) we obtain the $n-1$ equations

$$3a_i h_i^2 + 2b_i h_i + c_i = c_{i+1}, \quad i = 0 \text{ to } n-2$$

From (5.4) and (5.7) we obtain the $n-1$ equations

$$3a_i h_i + b_i = b_{i+1} \quad i = 0 \text{ to } n-2$$

It is will be useful to include a definition of the coefficient b_n, which is now missing. (The index of the cubic polynomial coefficients only go up to $n-1$.) We simply extend (5.11) up to $i=n-1$ and so write

$$3a_{n-1}h_{n-1}+b_{n-1}=b_n$$

which can be viewed as a definition of b_n.

We now proceed to eliminate the sets of a - and c-coefficients (with the d-coefficients already eliminated in (5.8)) to find a system of linear equations for the b-coefficients. From (5.11) and (5.12), we can solve for the na-coefficients to find

$$a_i=\frac{1}{3h_i}\left(b_{i+1}-b_i\right), \quad i=0 \text{ to } n-1$$

From (5.9), we can solve for the n c-coefficients as follows:

$$\begin{aligned} c_i &= \frac{1}{h_i}\left(f_i-a_ih_i^3-b_ih_i^2\right) \\ &= \frac{1}{h_i}\left(f_i-\frac{1}{3h_i}\left(b_{i+1}-b_i\right)h_i^3-b_ih_i^2\right) \\ &= \frac{f_i}{h_i}-\frac{1}{3}h_i\left(b_{i+1}+2b_i\right), \quad i=0 \text{ to } n-1. \end{aligned}$$

We can now find an equation for the b-coefficients by substituting (5.8), (5.13) and (5.14) into (5.10) :

$$\begin{aligned} 3\left(\frac{1}{3h_i}\left(b_{i+1}-b_i\right)\right)h_i^2+2b_ih_i+\left(\frac{f_i}{h_i}-\frac{1}{3}h_i\left(b_{i+1}+2b_i\right)\right) \\ =\left(\frac{f_{i+1}}{h_{i+1}}-\frac{1}{3}h_{i+1}\left(b_{i+2}+2b_{i+1}\right)\right) \end{aligned}$$

which simplifies to

$$\frac{1}{3}h_ib_i+\frac{2}{3}\left(h_i+h_{i+1}\right)b_{i+1}+\frac{1}{3}h_{i+1}b_{i+2}=\frac{f_{i+1}}{h_{i+1}}-\frac{f_i}{h_i}$$

an equation that is valid for $i=0$ to $n-2$. Therefore, (5.15) represent $n-1$ equations for the $n+1$ unknown b-coefficients. Accordingly, we write the matrix equa- tion for the b-coefficients, leaving the first and last row absent, as

$$\begin{pmatrix} \cdots & \cdots & \cdots & \cdots & \text{missing} & \cdots & \cdots \\ \frac{1}{3}h_0 & \frac{2}{3}\left(h_0+h_1\right) & \frac{1}{3}h_1 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & \frac{1}{3}h_{n-2} & \frac{2}{3}\left(h_{n-2}+h_{n-1}\right) & \frac{1}{3}h_{n-1} \\ \cdots & \cdots & \cdots & \cdots & \text{missing} & \cdots & \cdots \end{pmatrix}$$

$$\begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \\ b_n \end{pmatrix} \begin{pmatrix} \text{missing} \\ \frac{f_1}{h_1}-\frac{f_0}{h_0} \\ \vdots \\ \frac{f_{n-1}}{h_{n-1}}-\frac{f_{n-2}}{h_{n-2}} \\ \text{missing} \end{pmatrix}$$

Once the missing first and last equations are specified, the matrix equation for the b-coefficients can be solved by Gaussian elimination. And once the b-coefficients are determined, the a - and c-coefficients can also be determined from (5.13) and (5.14), with the d-coefficients already known from (5.8). The piecewise cubic polynomials, then, are known and $g(x)$ can be used for interpolation to any value x satisfying $x_0 \le x \le x_n$

The missing first and last equations can be specified in several ways, and here we show the two ways that are allowed by the MATLAB function spline.m. The first way should be used when the derivative $g'(x)$ is known at the endpoints x_0 and x_n; that is, suppose we know the values of α and β such that

$$g_0'\left(x_0\right)=\alpha, \quad g_{n-1}'\left(x_n\right)=\beta.$$

From the known value of α, and using (5.6) and (5.14), we have

$$\begin{aligned} \alpha &= c_0 \\ &= \frac{f_0}{\cdot}-\frac{1}{\cdot}h_0\left(b_1+2b_0\right) \end{aligned}$$

$h_0 \quad 3$

Therefore, the missing first equation is determined to be

$$\frac{2}{3}h_0 b_0 + \frac{1}{3}h_0 b_1 = \frac{f_0}{h_0} - \alpha.$$

From the known value of β, and using $(5.6), (5.13)$, and (5.14), we have

$$\begin{aligned}\beta &= 3a_{n-1}h_{n-1}^2 + 2b_{n-1}h_{n-1} + c_{n-1} \\ &= 3\left(\frac{1}{3h_{n-1}}\left(b_n - b_{n-1}\right)\right)h_{n-1}^2 + 2b_{n-1}h_{n-1} + \left(\frac{f_{n-1}}{h_{n-1}} - \frac{1}{3}h_{n-1}\left(b_n + 2b_{n-1}\right)\right),\end{aligned}$$

which simplifies to

$$\frac{1}{3}h_{n-1}b_{n-1} + \frac{2}{3}h_{n-1}b_n = \beta - \frac{f_{n-1}}{h_{n-1}}$$

to be used as the missing last equation.

The second way of specifying the missing first and last equations is called the not- a-knot condition, which assumes that

$$g_0(x) = g_1(x), \quad g_{n-2}(x) = g_{n-1}(x)$$

Considering the first of these equations, from (5.5) we have

$$\begin{aligned}a_0\left(x - x_0\right)^3 + b_0\left(x - x_0\right)^2 + c_0(x - x_0) + d_0 \\ = a_1\left(x - x_1\right)^3 + b_1\left(x - x_1\right)^2 + c_1\left(x - x_1\right) + d_1\end{aligned}$$

Now two cubic polynomials can be proven to be identical if at some value of x, the polynomials and their first three derivatives are identical. Our conditions of continuity at $x = x_1$ already require that at this value of x these two polynomials and their first two derivatives are identical. The polynomials themselves will be identical, then, if their third derivatives are also identical at $x = x_1$, or if

$$a_0 = a_1$$

From (5.13), we have

$$\frac{1}{3h_0}\left(b_1 - b_0\right) = \frac{1}{3h_1}\left(b_2 - b_1\right)$$

or after simplification

$$h_1 b_0 - \left(h_0 + h_1\right)b_1 + h_0 b_2 = 0$$

which provides us our missing first equation. A similar argument at $x = x_n - 1$ also provides us with our last equation,

$$h_{n-1}b_{n-2} - \left(h_{n-2} + h_{n-1}\right)b_{n-1} + h_{n-2}b_n = 0.$$

The MATLAB subroutines spline.m and ppval.m can be used for cubic spline interpolation (see also interp1.m). I will illustrate these routines in class and post sample code on the course web site.

This page titled 5.3: Cubic Spline Interpolation is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

5.4: Multidimensional Interpolation

Suppose we are interpolating the value of a function of two variables,

$$z = f(x, y)$$

The known values are given by

$$z_{ij} = f\left(x_i, y_j\right)$$

with $i = 0, 1, \ldots, n$ and $j = 0, 1, \ldots, n$. Note that the (x, y) points at which $f(x, y)$ are known lie on a grid in the $x - y$ plane.

Let $z = g(x, y)$ be the interpolating function, satisfying $z_{ij} = g\left(x_i, y_j\right)$. A twodimensional interpolation to find the value of g at the point (x, y) may be done by first performing $n + 1$ one-dimensional interpolations in y to find the value of g at the $n + 1$ points $\left(x_0, y\right), \left(x_1, y\right), \ldots, \left(x_n, y\right)$, followed by a single one-dimensional interpolation in x to find the value of g at (x, y).

In other words, two-dimensional interpolation on a grid of dimension $(n + 1) \times (n + 1)$ is done by first performing $n + 1$ one-dimensional interpolations to the value y followed by a single one-dimensional interpolation to the value x. Twodimensional interpolation can be generalized to higher dimensions. The MATLAB functions to perform two-and three-dimensional interpolation are interp2.m and interp3.m.

This page titled 5.4: Multidimensional Interpolation is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

CHAPTER OVERVIEW

6: Integration

We want to construct numerical algorithms that can perform definite integrals of the form

$$I = \int_a^b f(x)dx$$

Calculating these definite integrals numerically is called numerical integration, numerical quadrature, or more simply quadrature.

6.1: Elementary Formulas
6.2: Composite Rules
6.3: Local Versus Global Error
6.4: Adaptive Integration

This page titled 6: Integration is shared under a CC BY 1.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov.

6.1: Elementary Formulas

We first consider integration from 0 to h, with h small, to serve as the building blocks for integration over larger domains. We here define I_h as the following integral:

$$I_h = \int_0^h f(x)dx.$$

To perform this integral, we consider a Taylor series expansion of $f(x)$ about the value $x = h/2$:

$$f(x) = f(h/2) + (x-h/2)f'(h/2) + \frac{(x-h/2)^2}{2}f''(h/2) + \frac{(x-h/2)^3}{6}f'''(h/2) + \frac{(x-h/2)^4}{24}f''''(h/2) + ...$$

6.1.1. Midpoint rule

The midpoint rule makes use of only the first term in the Taylor series expansion. Here, we will determine the error in this approximation. Integrating,

$$I_h = hf(h/2) + \int_0^h \left((x-h/2)f'(h/2) + \frac{(x-h/2)^2}{2}f''(h/2) + \frac{(x-h/2)^3}{6}f'''(h/2) + \frac{(x-h/2)^4}{24}f''''(h/2) + ...\right)dx.$$

Changing variables by letting $y = x - h/2$ and $dy = dx$, and simplifying the integral depending on whether the integrand is even or odd, we have

$$\begin{aligned} I_h =& hf(h/2) \\ &+ \int_{-h/2}^{h/2} \left(yf'(h/2) + \frac{y^2}{2}f''(h/2) + \frac{y^3}{6}f'''(h/2) + \frac{y^4}{24}f''''(h/2) + ...\right)dy \\ =& hf(h/2) + \int_0^{h/2} \left(y^2 f''(h/2) + \frac{y^4}{12}f''''(h/2) + ...\right)dy \end{aligned}$$

The integrals that we need here are

$$\int_0^{\frac{h}{2}} y^2 dy = \frac{h^3}{24}, \quad \int_0^{\frac{h}{2}} y^4 dy = \frac{h^5}{160}$$

Therefore,

$$I_h = hf(h/2) + \frac{h^3}{24}f''(h/2) + \frac{h^5}{1920}f''''(h/2) + ...$$

6.1.2. Trapezoidal rule

From the Taylor series expansion of $f(x)$ about $x = h/2$, we have

$$f(0) = f(h/2) - \frac{h}{2}f'(h/2) + \frac{h^2}{8}f''(h/2) - \frac{h^3}{48}f'''(h/2) + \frac{h^4}{384}f''''(h/2) + ...,$$

and

$$f(h) = f(h/2) + \frac{h}{2}f'(h/2) + \frac{h^2}{8}f''(h/2) + \frac{h^3}{48}f'''(h/2) + \frac{h^4}{384}f''''(h/2) + ...$$

Adding and multiplying by $h/2$ we obtain

$$\frac{h}{2}(f(0) + f(h)) = hf(h/2) + \frac{h^3}{8}f''(h/2) + \frac{h^5}{384}f''''(h/2) + ...$$

We now substitute for the first term on the right-hand-side using the midpoint rule formula:

$$\tfrac{h}{2}(f(0) + f(h)) = \left(I_h - \tfrac{h^3}{24}f''(h/2) - \tfrac{h^5}{1920}f''''(h/2)\right)$$

$$+\tfrac{h^5}{8} f''(h/2) + \tfrac{h^5}{384} f''''(h/2) + \dots ,$$

and solving for I_h, we find

$$I_h = \frac{h}{2}(f(0) + f(h)) - \frac{h^3}{12} f''(h/2) - \frac{h^5}{480} f''''(h/2) + \dots$$

6.1.3. Simpson's rule

To obtain Simpson's rule, we combine the midpoint and trapezoidal rule to eliminate the error term proportional to h^3. Multiplying (6.3) by two and adding to (6.4), we obtain

$$3I_h = h\left(2f(h/2) + \frac{1}{2}(f(0) + f(h))\right) + h^5\left(\frac{2}{1920} - \frac{1}{480}\right) f''''(h/2) + \dots ,$$

or

$$I_h = \frac{h}{6}(f(0) + 4f(h/2) + f(h)) - \frac{h^5}{2880} f''''(h/2) + \dots .$$

Usually, Simpson's rule is written by considering the three consecutive points $0, h$ and $2h$. Substituting $h \to 2h$, we obtain the standard result

$$I_{2h} = \frac{h}{3}(f(0) + 4f(h) + f(2h)) - \frac{h^5}{90} f''''(h) + \dots$$

This page titled 6.1: Elementary Formulas is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

6.2: Composite Rules

We now use our elementary formulas obtained for (6.2) to perform the integral given by (6.1)

6.2.1. Trapezoidal rule

We suppose that the function $f(x)$ is known at the $n+1$ points labeled as $x_0, x_1, \dots, x_n$, with the endpoints given by $x_0 = a$ and $x_n = b$. Define

$$f_i = f(x_i), \quad h_i = x_{i+1} - x_i$$

Then the integral of (6.1) may be decomposed as

$$\begin{aligned}\int_a^b f(x)dx &= \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} f(x)dx \\ &= \sum_{i=0}^{n-1} \int_0^{h_i} f(x_i + s)\, ds\end{aligned}$$

where the last equality arises from the change-of-variables $s = x - x_i$. Applying the trapezoidal rule to the integral, we have

$$\int_a^b f(x)dx = \sum_{i=0}^{n-1} \frac{h_i}{2}(f_i + f_{i+1})$$

If the points are not evenly spaced, say because the data are experimental values, then the h_i may differ for each value of i and (6.6) is to be used directly.

However, if the points are evenly spaced, say because $f(x)$ can be computed, we have $h_i = h$, independent of i. We can then define

$$x_i = a + ih, \quad i = 0, 1, \dots, n$$

and since the end point b satisfies $b = a + nh$, we have

$$h = \frac{b-a}{n}$$

The composite trapezoidal rule for evenly space points then becomes

$$\begin{aligned}\int_a^b f(x)dx &= \frac{h}{2} \sum_{i=0}^{n-1} (f_i + f_{i+1}) \\ &= \frac{h}{2}(f_0 + 2f_1 + \cdots + 2f_{n-1} + f_n)\end{aligned}$$

The first and last terms have a multiple of one; all other terms have a multiple of two; and the entire sum is multiplied by $h/2$.

6.2.2. Simpson's rule

We here consider the composite Simpson's rule for evenly space points. We apply Simpson's rule over intervals of $2h$, starting from a and ending at b :

$$\begin{aligned}\int_a^b f(x)dx = \frac{h}{3}(f_0 + 4f_1 + f_2) + \frac{h}{3}(f_2 + 4f_3 + f_4) + \dots \\ + \frac{h}{3}(f_{n-2} + 4f_{n-1} + f_n)\end{aligned}$$

Note that n must be even for this scheme to work. Combining terms, we have

$$\int_a^b f(x)dx = \frac{h}{3}(f_0 + 4f_1 + 2f_2 + 4f_3 + 2f_4 + \cdots + 4f_{n-1} + f_n)$$

The first and last terms have a multiple of one; the even indexed terms have a multiple of 2; the odd indexed terms have a multiple of 4; and the entire sum is multiplied by $h/3$.

This page titled 6.2: Composite Rules is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

6.3: Local Versus Global Error

Consider the elementary formula (6.4) for the trapezoidal rule, written in the form

$$\int_0^h f(x)dx = \frac{h}{2}(f(0)+f(h)) - \frac{h^3}{12}f''(\xi)$$

where ξ is some value satisfying $0 \le \xi \le h$, and we have used Taylor's theorem with the mean-value form of the remainder. We can also represent the remainder as

$$-\frac{h^3}{12}f''(\xi) = \mathrm{O}\left(h^3\right)$$

where $\mathrm{O}\left(h^3\right)$ signifies that when h is small, halving of the grid spacing h decreases the error in the elementary trapezoidal rule by a factor of eight. We call the terms represented by $\mathrm{O}\left(h^3\right)$ the Local Error.

More important is the Global Error which is obtained from the composite formula (6.7) for the trapezoidal rule. Putting in the remainder terms, we have

$$\int_a^b f(x)dx = \frac{h}{2}\left(f_0 + 2f_1 + \cdots + 2f_{n-1} + f_n\right) - \frac{h^3}{12}\sum_{i=0}^{n-1} f''\left(\xi_i\right)$$

where ξ_i are values satisfying $x_i \le \xi_i \le x_{i+1}$. The remainder can be rewritten as

$$-\frac{h^3}{12}\sum_{i=0}^{n-1} f''\left(\xi_i\right) = -\frac{nh^3}{12}\left\langle f''\left(\xi_i\right)\right\rangle$$

where $\left\langle f''\left(\xi_i\right)\right\rangle$ is the average value of all the $f''\left(\xi_i\right)'$ s. Now,

$$n = \frac{b-a}{h}$$

so that the error term becomes

$$\begin{aligned} -\frac{nh^3}{12}\left\langle f''\left(\xi_i\right)\right\rangle &= -\frac{(b-a)h^2}{12}\left\langle f''\left(\xi_i\right)\right\rangle \\ &= \mathrm{O}\left(h^2\right) \end{aligned}$$

Therefore, the global error is $\mathrm{O}\left(h^2\right)$. That is, a halving of the grid spacing only decreases the global error by a factor of four.

Similarly, Simpson's rule has a local error of $\mathrm{O}\left(h^5\right)$ and a global error of $\mathrm{O}\left(h^4\right)$.

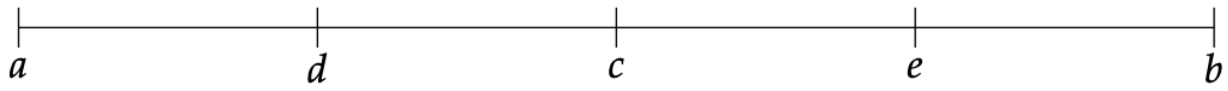


Figure 6.1: Adaptive Simpson quadrature: Level 1.

This page titled 6.3: Local Versus Global Error is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

6.4: Adaptive Integration

The useful MATLAB function quad.m performs numerical integration using adaptive Simpson quadrature. The idea is to let the computation itself decide on the grid size required to achieve a certain level of accuracy. Moreover, the grid size need not be the same over the entire region of integration.

We begin the adaptive integration at what is called Level 1. The uniformly spaced points at which the function $f(x)$ is to be evaluated are shown in Fig. 6.1. The distance between the points a and b is taken to be $2h$, so that

$$h = \frac{b-a}{2}$$

Integration using Simpson's rule (6.5) with grid size h yields

$$I = \frac{h}{3}(f(a) + 4f(c) + f(b)) - \frac{h^5}{90}f''''(\xi)$$

where ξ is some value satisfying $a \le \xi \le b$.

Integration using Simpson's rule twice with grid size $h/2$ yields

$$I = \frac{h}{6}(f(a) + 4f(d) + 2f(c) + 4f(e) + f(b)) - \frac{(h/2)^5}{90}f''''(\xi_l) - \frac{(h/2)^5}{90}f''''(\xi_r)$$

with ξ_l and ξ_r some values satisfying $a \le \xi_l \le c$ and $c \le \xi_r \le b$.

We now define

$$\begin{aligned}
S_1 &= \frac{h}{3}(f(a) + 4f(c) + f(b)) \\
S_2 &= \frac{h}{6}(f(a) + 4f(d) + 2f(c) + 4f(e) + f(b)), \\
E_1 &= -\frac{h^5}{90}f''''(\xi), \\
E_2 &= -\frac{h^5}{2^5 \cdot 90}\left(f''''(\xi_l) + f''''(\xi_r)\right).
\end{aligned}$$

Now we ask whether S_2 is accurate enough, or must we further refine the calculation and go to Level 2? To answer this question, we make the simplifying approximation that all of the fourth-order derivatives of $f(x)$ in the error terms are equal; that is,

$$f''''(\xi) = f''''(\xi_l) = f''''(\xi_r) = C.$$

Then

$$\begin{aligned}
E_1 &= -\frac{h^5}{90}C \\
E_2 &= -\frac{h^5}{2^4 \cdot 90}C = \frac{1}{16}E_1
\end{aligned}$$

Then since

$$S_1 + E_1 = S_2 + E_2$$

and

$$E_1 = 16E_2$$

we have for our estimate for the error term E_2,

$$E_2 = \frac{1}{15}(S_2 - S_1)$$

Therefore, given some specific value of the tolerance tol, if

$$\left|\frac{1}{15}(S_2 - S_1)\right| < \text{ tol}$$

then we can accept S_2 as I. If the tolerance is not achieved for I, then we proceed to Level 2.

The computation at Level 2 further divides the integration interval from a to b into the two integration intervals a to c and c to b, and proceeds with the above procedure independently on both halves. Integration can be stopped on either half provided the tolerance is less than tol/2 (since the sum of both errors must be less than tol). Otherwise, either half can proceed to Level 3, and so on.

As a side note, the two values of I given above (for integration with step size h and $h/2$) can be combined to give a more accurate value for I given by

$$I = \frac{16S_2 - S_1}{15} + \mathrm{O}\left(h^7\right)$$

where the error terms of $\mathrm{O}\left(h^5\right)$ approximately cancel. This free lunch, so to speak, is called Richardson's extrapolation.

This page titled 6.4: Adaptive Integration is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

CHAPTER OVERVIEW

7: Ordinary Differential Equations

We now discuss the numerical solution of ordinary differential equations. These include the initial value problem, the boundary value problem, and the eigenvalue problem. Before proceeding to the development of numerical methods, we review the analytical solution of some classic equations.

7.1: Examples of Analytical Solutions

7.2: Numerical Methods - Initial Value Problem

7.3: Numerical Methods - Boundary Value Problem

7.4: Numerical Methods - Eigenvalue Problem

This page titled 7: Ordinary Differential Equations is shared under a CC BY 1.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov.

7.1: Examples of Analytical Solutions

7.1.1. Initial value problem

One classic initial value problem is the RC circuit. With R the resistor and C the capacitor, the differential equation for the charge q on the capacitor is given by

$$R\frac{dq}{dt} + \frac{q}{C} = 0.$$

If we consider the physical problem of a charged capacitor connected in a closed circuit to a resistor, then the initial condition is $q(0) = q_0$, where q_0 is the initial charge on the capacitor.

The differential equation (7.1) is separable, and separating and integrating from time $t = 0$ to t yields

$$\int_{q_0}^{q} \frac{dq}{q} = -\frac{1}{RC}\int_0^t dt$$

which can be integrated and solved for $q = q(t)$:

$$q(t) = q_0 e^{-t/RC}.$$

The classic second-order initial value problem is the RLC circuit, with differential equation

$$L\frac{d^2q}{dt^2} + R\frac{dq}{dt} + \frac{q}{C} = 0.$$

Here, a charged capacitor is connected to a closed circuit, and the initial conditions satisfy

$$q(0) = q_0, \quad \frac{dq}{dt}(0) = 0$$

The solution is obtained for the second-order equation by the ansatz

$$q(t) = e^{rt}$$

which results in the following so-called characteristic equation for r :

$$Lr^2 + Rr + \frac{1}{C} = 0$$

If the two solutions for r are distinct and real, then the two found exponential solutions can be multiplied by constants and added to form a general solution. The constants can then be determined by requiring the general solution to satisfy the two initial conditions. If the roots of the characteristic equation are complex or degenerate, a general solution to the differential equation can also be found.

7.1.2. Boundary value problems

The dimensionless equation for the temperature $y = y(x)$ along a linear heatconducting rod of length unity, and with an applied external heat source $f(x)$, is given by the differential equation

$$-\frac{d^2y}{dx^2} = f(x)$$

with $0 \le x \le 1$. Boundary conditions are usually prescribed at the end points of the rod, and here we assume that the temperature at both ends are maintained at zero so that

$$y(0) = 0, \quad y(1) = 0$$

The assignment of boundary conditions at two separate points is called a twopoint boundary value problem, in contrast to the initial value problem where the boundary conditions are prescribed at only a single point. Two-point boundary value problems typically require a more sophisticated algorithm for a numerical solution than initial value problems.

Here, the solution of (7.2) can proceed by integration once $f(x)$ is specified. We assume that

$$f(x) = x(1 - x)$$

so that the maximum of the heat source occurs in the center of the rod, and goes to zero at the ends.

The differential equation can then be written as

$$\frac{d^2y}{dx^2} = -x(1-x)$$

The first integration results in

$$\begin{aligned}\frac{dy}{dx} &= \int (x^2 - x)\, dx \\ &= \frac{x^3}{3} - \frac{x^2}{2} + c_1\end{aligned}$$

where c_1 is the first integration constant. Integrating again,

$$\begin{aligned}y(x) &= \int \left(\frac{x^3}{3} - \frac{x^2}{2} + c_1\right) dx \\ &= \frac{x^4}{12} - \frac{x^3}{6} + c_1 x + c_2\end{aligned}$$

where c_2 is the second integration constant. The two integration constants are determined by the boundary conditions. At $x = 0$, we have

$$0 = c_2$$

and at $x = 1$, we have

$$0 = \frac{1}{12} - \frac{1}{6} + c_1$$

so that $c_1 = 1/12$. Our solution is therefore

$$\begin{aligned}y(x) &= \frac{x^4}{12} - \frac{x^3}{6} + \frac{x}{12} \\ &= \frac{1}{12} x(1-x)\left(1 + x - x^2\right)\end{aligned}$$

The temperature of the rod is maximum at $x = 1/2$ and goes smoothly to zero at the ends.

7.1.3. Eigenvalue problem

The classic eigenvalue problem obtained by solving the wave equation by separation of variables is given by

$$\frac{d^2y}{dx^2} + \lambda^2 y = 0$$

with the two-point boundary conditions $y(0) = 0$ and $y(1) = 0$. Notice that $y(x) = 0$ satisfies both the differential equation and the boundary conditions. Other nonzero solutions for $y = y(x)$ are possible only for certain discrete values of λ. These values of λ are called the eigenvalues of the differential equation.

We proceed by first finding the general solution to the differential equation. It is easy to see that this solution is

$$y(x) = A\cos\lambda x + B\sin\lambda x$$

Imposing the first boundary condition at $x = 0$, we obtain

$$A = 0$$

The second boundary condition at $x = 1$ results in

$$B\sin\lambda = 0$$

Since we are searching for a solution where $y = y(x)$ is not identically zero, we must have

$$\lambda = \pi, 2\pi, 3\pi, \ldots$$

The corresponding negative values of λ are also solutions, but their inclusion only changes the corresponding values of the unknown B constant. A linear superposition of all the solutions results in the general solution

$$y(x) = \sum_{n=1}^{\infty} B_n \sin n\pi x.$$

For each eigenvalue $n\pi$, we say there is a corresponding eigenfunction $\sin n\pi x$. When the differential equation can not be solved analytically, a numerical method should be able to solve for both the eigenvalues and eigenfunctions.

This page titled 7.1: Examples of Analytical Solutions is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

7.2: Numerical Methods - Initial Value Problem

We begin with the simple Euler method, then discuss the more sophisticated RungeKutta methods, and conclude with the Runge-Kutta-Fehlberg method, as implemented in the MATLAB function ode45.m. Our differential equations are for $x = x(t)$, where the time t is the independent variable, and we will make use of the notation $\dot{x} = dx/dt$. This notation is still widely used by physicists and descends directly from the notation originally used by Newton.

7.2.1. Euler method

The Euler method is the most straightforward method to integrate a differential equation. Consider the first-order differential equation

$$\dot{x} = f(t, x),$$

with the initial condition $x(0) = x_0$. Define $t_n = n\Delta t$ and $x_n = x\left(t_n\right)$. A Taylor series expansion of x_{n+1} results in

$$\begin{aligned} x_{n+1} &= x\left(t_n + \Delta t\right) \\ &= x\left(t_n\right) + \Delta t \dot{x}\left(t_n\right) + \mathrm{O}\left(\Delta t^2\right) \\ &= x\left(t_n\right) + \Delta t f\left(t_n, x_n\right) + \mathrm{O}\left(\Delta t^2\right) \end{aligned}$$

The Euler Method is therefore written as

$$x_{n+1} = x\left(t_n\right) + \Delta t f\left(t_n, x_n\right)$$

We say that the Euler method steps forward in time using a time-step Δt, starting from the initial value $x_0 = x(0)$. The local error of the Euler Method is $\mathrm{O}\left(\Delta t^2\right)$. The global error, however, incurred when integrating to a time T, is a factor of $1/\Delta t$ larger and is given by $\mathrm{O}(\Delta t)$. It is therefore customary to call the Euler Method a first-order method.

7.2.2. Modified Euler method

This method is of a type that is called a predictor-corrector method. It is also the first of what are Runge-Kutta methods. As before, we want to solve (7.3). The idea is to average the value of $\dot{x}$ at the beginning and end of the time step. That is, we would like to modify the Euler method and write

$$x_{n+1} = x_n + \frac{1}{2}\Delta t\left(f\left(t_n, x_n\right) + f\left(t_n + \Delta t, x_{n+1}\right)\right)$$

The obvious problem with this formula is that the unknown value x_{n+1} appears on the right-hand-side. We can, however, estimate this value, in what is called the predictor step. For the predictor step, we use the Euler method to find

$$x_{n+1}^p = x_n + \Delta t f\left(t_n, x_n\right)$$

The corrector step then becomes

$$x_{n+1} = x_n + \frac{1}{2}\Delta t\left(f\left(t_n, x_n\right) + f\left(t_n + \Delta t, x_{n+1}^p\right)\right)$$

The Modified Euler Method can be rewritten in the following form that we will later identify as a Runge-Kutta method:

$$\begin{aligned} k_1 &= \Delta t f\left(t_n, x_n\right) \\ k_2 &= \Delta t f\left(t_n + \Delta t, x_n + k_1\right) \\ x_{n+1} &= x_n + \frac{1}{2}\left(k_1 + k_2\right) \end{aligned}$$

7.2.3. Second-order Runge-Kutta methods

We now derive all second-order Runge-Kutta methods. Higher-order methods can be similarly derived, but require substantially more algebra.

We consider the differential equation given by (7.3). A general second-order Runge-Kutta method may be written in the form

$$\begin{aligned} k_1 &= \Delta t f\left(t_n, x_n\right) \\ k_2 &= \Delta t f\left(t_n + \alpha \Delta t, x_n + \beta k_1\right) \\ x_{n+1} &= x_n + a k_1 + b k_2 \end{aligned}$$

with α, β, a and b constants that define the particular second-order Runge-Kutta method. These constants are to be constrained by setting the local error of the second-order Runge-Kutta method to be $\mathrm{O}\left(\Delta t^3\right)$. Intuitively, we might guess that two of the constraints will be $a+b=1$ and $\alpha=\beta$.

We compute the Taylor series of x_{n+1} directly, and from the Runge-Kutta method, and require them to be the same to order Δt^2. First, we compute the Taylor series of x_{n+1}. We have

$$\begin{aligned} x_{n+1} &= x\left(t_n+\Delta t\right) \\ &= x\left(t_n\right)+\Delta t \dot{x}\left(t_n\right)+\frac{1}{2}(\Delta t)^2 \ddot{x}\left(t_n\right)+\mathrm{O}\left(\Delta t^3\right) \end{aligned}$$

Now,

$$\dot{x}\left(t_n\right)=f\left(t_n, x_n\right).$$

The second derivative is more complicated and requires partial derivatives. We have

$$\begin{aligned} \ddot{x}\left(t_n\right) &= \frac{d}{dt} f(t, x(t))\Big]_{t=t_n} \\ &= f_t\left(t_n, x_n\right)+\dot{x}\left(t_n\right) f_x\left(t_n, x_n\right) \\ &= f_t\left(t_n, x_n\right)+f\left(t_n, x_n\right) f_x\left(t_n, x_n\right) \end{aligned}$$

Therefore,

$$x_{n+1}=x_n+\Delta t f\left(t_n, x_n\right)+\frac{1}{2}(\Delta t)^2\left(f_t\left(t_n, x_n\right)+f\left(t_n, x_n\right) f_x\left(t_n, x_n\right)\right)$$

Second, we compute x_{n+1} from the Runge-Kutta method given by (7.5). Substituting in k_1 and k_2, we have

$$x_{n+1}=x_n+a \Delta t f\left(t_n, x_n\right)+b \Delta t f\left(t_n+\alpha \Delta t, x_n+\beta \Delta t f\left(t_n, x_n\right)\right).$$

We Taylor series expand using

$$\begin{aligned} & f\left(t_n+\alpha \Delta t, x_n+\beta \Delta t f\left(t_n, x_n\right)\right) \\ &= f\left(t_n, x_n\right)+\alpha \Delta t f_t\left(t_n, x_n\right)+\beta \Delta t f\left(t_n, x_n\right) f_x\left(t_n, x_n\right)+\mathrm{O}\left(\Delta t^2\right) \end{aligned}$$

The Runge-Kutta formula is therefore

$$\begin{aligned} x_{n+1} = x_n &+(a+b) \Delta t f\left(t_n, x_n\right) \\ &+(\Delta t)^2\left(\alpha b f_t\left(t_n, x_n\right)+\beta b f\left(t_n, x_n\right) f_x\left(t_n, x_n\right)\right)+\mathrm{O}\left(\Delta t^3\right) \end{aligned}$$

Comparing (7.6) and (7.7), we find

$$\begin{aligned} a+b &= 1 \\ \alpha b &= 1/2 \\ \beta b &= 1/2 \end{aligned}$$

There are three equations for four parameters, and there exists a family of secondorder Runge-Kutta methods.

The Modified Euler Method given by (7.4) corresponds to $\alpha=\beta=1$ and $a=b=1/2$. Another second-order Runge-Kutta method, called the Midpoint Method, corresponds to $\alpha=\beta=1/2, a=0$ and $b=1$. This method is written as

$$\begin{aligned} k_1 &= \Delta t f\left(t_n, x_n\right) \\ k_2 &= \Delta t f\left(t_n+\frac{1}{2} \Delta t, x_n+\frac{1}{2} k_1\right) \\ x_{n+1} &= x_n+k_2 \end{aligned}$$

7.2.4. Higher-order Runge-Kutta methods

The general second-order Runge-Kutta method was given by (7.5). The general form of the third-order method is given by

$$\begin{aligned} k_1 &= \Delta t f\left(t_n, x_n\right) \\ k_2 &= \Delta t f\left(t_n+\alpha \Delta t, x_n+\beta k_1\right) \\ k_3 &= \Delta t f\left(t_n+\gamma \Delta t, x_n+\delta k_1+\epsilon k_2\right) \\ x_{n+1} &= x_n+a k_1+b k_2+c k_3 \end{aligned}$$

The following constraints on the constants can be guessed: $\alpha = \beta, \gamma = \delta + \epsilon$, and $a + b + c = 1$. Remaining constraints need to be derived.

The fourth-order method has a k_1, k_2, k_3 and k_4. The fifth-order method requires up to k_6. The table below gives the order of the method and the number of stages required.

order	2	3	4	5	6	7	8
minimum # stages	2	3	4	6	7	9	11

Because of the jump in the number of stages required between the fourth-order and fifth-order method, the fourth-order Runge-Kutta method has some appeal. The general fourth-order method starts with 13 constants, and one then finds 11 constraints. A particularly simple fourth-order method that has been widely used is given by

$$\begin{aligned} k_1 &= \Delta t f\left(t_n, x_n\right) \\ k_2 &= \Delta t f\left(t_n + \frac{1}{2}\Delta t, x_n + \frac{1}{2}k_1\right) \\ k_3 &= \Delta t f\left(t_n + \frac{1}{2}\Delta t, x_n + \frac{1}{2}k_2\right) \\ k_4 &= \Delta t f\left(t_n + \Delta t, x_n + k_3\right) \\ x_{n+1} &= x_n + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \end{aligned}$$

7.2.5. Adaptive Runge-Kutta Methods

As in adaptive integration, it is useful to devise an ode integrator that automatically finds the appropriate Δt. The Dormand-Prince Method, which is implemented in MATLAB's ode45.m, finds the appropriate step size by comparing the results of a fifth-order and fourth-order method. It requires six function evaluations per time step, and constructs both a fifth-order and a fourth-order method from these function evaluations.

Suppose the fifth-order method finds x_{n+1} with local error $\mathrm{O}\left(\Delta t^6\right)$, and the fourth-order method finds x'_{n+1} with local error $\mathrm{O}\left(\Delta t^5\right)$. Let ε be the desired error tolerance of the method, and let e be the actual error. We can estimate e from the difference between the fifth-and fourth-order methods; that is,

$$e = \left|x_{n+1} - x'_{n+1}\right|$$

Now e is of $\mathrm{O}\left(\Delta t^5\right)$, where Δt is the step size taken. Let $\Delta\tau$ be the estimated step size required to get the desired error ε. Then we have

$$e/\varepsilon = (\Delta t)^5/(\Delta\tau)^5$$

or solving for $\Delta\tau$,

$$\Delta\tau = \Delta t\left(\frac{\varepsilon}{e}\right)^{1/5}$$

On the one hand, if $e < \varepsilon$, then we accept x_{n+1} and do the next time step using the larger value of $\Delta\tau$. On the other hand, if $e > \varepsilon$, then we reject the integration step and redo the time step using the smaller value of $\Delta\tau$. In practice, one usually increases the time step slightly less and decreases the time step slightly more to prevent the waste of too many failed time steps.

7.2.6. System of differential equations

Our numerical methods can be easily adapted to solve higher-order differential equations, or equivalently, a system of differential equations. First, we show how a second-order differential equation can be reduced to two first-order equations. Consider

$$\ddot{x} = f(t, x, \dot{x}).$$

This second-order equation can be rewritten as two first-order equations by defining $u = \dot{x}$. We then have the system

$$\begin{aligned} \dot{x} &= u, \\ \dot{u} &= f(t, x, u). \end{aligned}$$

This trick also works for higher-order equation. For another example, the thirdorder equation

$$\ddot{x} = f(t, x, \dot{x}, \ddot{x}),$$

can be written as

$$\begin{aligned}\dot{x} &= u\\ \dot{u} &= v\\ \dot{v} &= f(t, x, u, v).\end{aligned}$$

Now, we show how to generalize Runge-Kutta methods to a system of differential equations. As an example, consider the following system of two odes,

$$\begin{aligned}\dot{x} &= f(t, x, y),\\ \dot{y} &= g(t, x, y),\end{aligned}$$

with the initial conditions $x(0) = x_0$ and $y(0) = y_0$. The generalization of the commonly used fourth-order Runge-Kutta method would be

$$\begin{aligned}k_1 &= \Delta t f\left(t_n, x_n, y_n\right)\\ l_1 &= \Delta t g\left(t_n, x_n, y_n\right)\\ k_2 &= \Delta t f\left(t_n + \frac{1}{2}\Delta t, x_n + \frac{1}{2}k_1, y_n + \frac{1}{2}l_1\right)\\ l_2 &= \Delta t g\left(t_n + \frac{1}{2}\Delta t, x_n + \frac{1}{2}k_1, y_n + \frac{1}{2}l_1\right)\\ y_3 &= \Delta t f\left(t_n + \frac{1}{2}\Delta t, x_n + \frac{1}{2}k_2, y_n + \frac{1}{2}l_2\right)\\ l_3 &= \Delta t g\left(t_n + \frac{1}{2}\Delta t, x_n + \frac{1}{2}k_2, y_n + \frac{1}{2}l_2\right)\\ y_{n+1} &= y_n + \frac{1}{6}\left(l_1 + 2l_2 + 2l_3 + l_4\right)\\ k_4 &= \Delta t f\left(t_n + \Delta t, x_n + k_3, y_n + l_3\right)\\ l_4 &= \Delta t g\left(t_n + \Delta t, x_n + k_3, y_n + l_3\right)\\ &= x_n + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)\\ &= \Delta\end{aligned}$$

This page titled 7.2: Numerical Methods - Initial Value Problem is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

7.3: Numerical Methods - Boundary Value Problem

7.3.1. Finite difference method

We consider first the differential equation

$$-\frac{d^2y}{dx^2} = f(x), \quad 0 \le x \le 1$$

with two-point boundary conditions

$$y(0) = A, \quad y(1) = B.$$

Equation (7.8) can be solved by quadrature, but here we will demonstrate a numerical solution using a finite difference method.

We begin by discussing how to numerically approximate derivatives. Consider the Taylor series approximation for $y(x+h)$ and $y(x-h)$, given by

$$y(x+h) = y(x) + hy'(x) + \frac{1}{2}h^2y''(x) + \frac{1}{6}h^3y'''(x) + \frac{1}{24}h^4y''''(x) + \dots,$$
$$y(x-h) = y(x) - hy'(x) + \frac{1}{2}h^2y''(x) - \frac{1}{6}h^3y'''(x) + \frac{1}{24}h^4y''''(x) + \dots$$

The standard definitions of the derivatives give the first-order approximations

$$y'(x) = \frac{y(x+h) - y(x)}{h} + \mathrm{O}(h)$$
$$y'(x) = \frac{y(x) - y(x-h)}{h} + \mathrm{O}(h)$$

The more widely-used second-order approximation is called the central difference approximation and is given by

$$y'(x) = \frac{y(x+h) - y(x-h)}{2h} + \mathrm{O}\left(h^2\right)$$

The finite difference approximation to the second derivative can be found from considering

$$y(x+h) + y(x-h) = 2y(x) + h^2y''(x) + \frac{1}{12}h^4y''''(x) + \dots,$$

from which we find

$$y''(x) = \frac{y(x+h) - 2y(x) + y(x-h)}{h^2} + \mathrm{O}\left(h^2\right).$$

Sometimes a second-order method is required for x on the boundaries of the domain. For a boundary point on the left, a second-order forward difference method requires the additional Taylor series

$$y(x+2h) = y(x) + 2hy'(x) + 2h^2y''(x) + \frac{4}{3}h^3y'''(x) + \dots$$

We combine the Taylor series for $y(x+h)$ and $y(x+2h)$ to eliminate the term proportional to h^2 :

$$y(x+2h) - 4y(x+h) = -3y(x) - 2hy'(x) + \mathrm{O}\left(\mathrm{h}^3\right)$$

Therefore,

$$y'(x) = \frac{-3y(x) + 4y(x+h) - y(x+2h)}{2h} + \mathrm{O}\left(h^2\right)$$

For a boundary point on the right, we send $h \to -h$ to find

$$y'(x) = \frac{3y(x) - 4y(x-h) + y(x-2h)}{2h} + \mathrm{O}\left(h^2\right)$$

We now write a finite difference scheme to solve (7.8). We discretize x by defining $x_i = ih, i = 0, 1, \dots, n+1$. Since $x_{n+1} = 1$, we have $h = 1/(n+1)$. The functions $y(x)$ and $f(x)$ are discretized as $y_i = y\left(x_i\right)$ and $f_i = f\left(x_i\right)$. The second-order differential equation (7.8) then becomes for the interior points of the domain

$$-y_{i-1} + 2y_i - y_{i+1} = h^2 f_i, \quad i = 1, 2, \ldots n,$$

with the boundary conditions $y_0 = A$ and $y_{n+1} = B$. We therefore have a linear system of equations to solve. The first and nth equation contain the boundary conditions and are given by

$$\begin{aligned} 2y_1 - y_2 &= h^2 f_1 + A \\ -y_{n-1} + 2y_n &= h^2 f_n + B \end{aligned}$$

The second and third equations, etc., are

$$\begin{aligned} -y_1 + 2y_2 - y_3 &= h^2 f_2 \\ -y_2 + 2y_3 - y_4 &= h^2 f_3 \end{aligned}$$

In matrix form, we have

$$\begin{pmatrix} 2 & -1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & \cdots & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \\ y_n \end{pmatrix} = \begin{pmatrix} h^2 f_1 + A \\ h^2 f_2 \\ h^2 f_3 \\ \vdots \\ h^2 f_{n-1} \\ h^2 f_n + B \end{pmatrix}$$

The matrix is tridiagonal, and a numerical solution by Guassian elimination can be done quickly. The matrix itself is easily constructed using the MATLAB function diag.m and ones.m. As excerpted from the MATLAB help page, the function call ones (m, n) returns an m-by-n matrix of ones, and the function call $\mathrm{diag}(\mathrm{v}, \mathrm{k})$, when v is a vector with n components, is a square matrix of order $\mathrm{n} + \mathrm{abs}(\mathrm{k})$ with the elements of v on the k-th diagonal: $k = 0$ is the main diagonal, $k > 0$ is above the main diagonal and $k < 0$ is below the main diagonal. The $n \times n$ matrix above can be constructed by the MATLAB code

$$M = \mathrm{diag}(-\ \mathrm{ones}\ (n-1, 1), -1) + \mathrm{diag}\,(2^*\ \mathrm{ones}\ (n, 1), 0) + \mathrm{diag}(-\ \mathrm{ones}\ (n-1, 1), 1);$$

The right-hand-side, provided f is a given n-by-1 vector, can be constructed by the MATLAB code

$$\mathrm{b} = \mathrm{h\hat{}2^*f}; \mathrm{b}(1) = \mathrm{b}(1) + \mathrm{A}; \mathrm{b}(\mathrm{n}) = \mathrm{b}(\mathrm{n}) + \mathrm{B}$$

and the solution for u is given by the MATLAB code

$$\mathrm{y} = \mathrm{M} \backslash \mathrm{b}$$

7.3.2. Shooting method

The finite difference method can solve linear odes. For a general ode of the form

$$\frac{d^2y}{dx^2} = f(x, y, dy/dx)$$

with $y(0) = A$ and $y(1) = B$, we use a shooting method. First, we formulate the ode as an initial value problem. We have

$$\begin{aligned} \frac{dy}{dx} &= z \\ \frac{dz}{dx} &= f(x, y, z) \end{aligned}$$

The initial condition $y(0) = A$ is known, but the second initial condition $z(0) = b$ is unknown. Our goal is to determine b such that $y(1) = B$.

In fact, this is a root-finding problem for an appropriately defined function. We define the function $F = F(b)$ such that

$$F(b) = y(1) - B$$

In other words, $F(b)$ is the difference between the value of $y(1)$ obtained from integrating the differential equations using the initial condition $z(0) = b$, and B. Our root-finding routine will want to solve $F(b) = 0$. (The method is called shooting because the slope of the solution curve for $y = y(x)$ at $x = 0$ is given by b, and the solution hits the value $y(1)$ at $x = 1$. This looks like pointing a gun and trying to shoot the target, which is B.)

To determine the value of b that solves $F(b) = 0$, we iterate using the Secant method, given by

$$b_{n+1} = b_n - F(b_n) \frac{b_n - b_{n-1}}{F(b_n) - F(b_{n-1})}$$

We need to start with two initial guesses for b, solving the ode for the two corresponding values of $y(1)$. Then the Secant Method will give us the next value of b to try, and we iterate until $|y(1) - B| <$ tol, where tol is some specified tolerance for the error

---

This page titled 7.3: Numerical Methods - Boundary Value Problem is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

7.4: Numerical Methods - Eigenvalue Problem

For illustrative purposes, we develop our numerical methods for what is perhaps the simplest eigenvalue ode. With $y = y(x)$ and $0 \leq x \leq 1$, this simple ode is given by

$$y'' + \lambda^2 y = 0 \tag{7.4.1}$$

To solve Equation 7.4.1 numerically, we will develop both a finite difference method and a shooting method. Furthermore, we will show how to solve (7.9) with homogeneous boundary conditions on either the function y or its derivative y'.

Finite difference method

We first consider solving (7.9) with the homogeneous boundary conditions $y(0) = y(1) = 0$. In this case, we have already shown that the eigenvalues of Equation 7.4.1 are given by $\lambda = \pi, 2\pi, 3\pi, ...$

With n interior points, we have $x_i = ih$ for $i = 0, \ldots, n+1$, and $h = 1/(n+1)$. Using the centered-finite-difference approximation for the second derivative, Equation 7.4.1 becomes

$$y_{i-1} - 2y_i + y_{i+1} = -h^2\lambda^2 y_i$$

Applying the boundary conditions $y_0 = y_{n+1} = 0$, the first equation with $i = 1$, and the last equation with $i = n$, are given by

$$\begin{aligned} -2y_1 + y_2 &= -h^2\lambda^2 y_1 \\ y_{n-1} - 2y_n &= -h^2\lambda^2 y_n \end{aligned}$$

The remaining $n-2$ equations are given by (7.10) for $i = 2, \ldots, n-1$ It is of interest to see how the solution develops with increasing n. The smallest possible value is $n = 1$, corresponding to a single interior point, and since $h = 1/2$ we have

$$-2y_1 = -\frac{1}{4}\lambda^2 y_1$$

so that $\lambda^2 = 8$, or $\lambda = 2\sqrt{2} = 2.8284$. This is to be compared to the first eigenvalue which is $\lambda = \pi$. When $n = 2$, we have $h = 1/3$, and the resulting two equations written in matrix form are given by

$$\begin{pmatrix} -2 & 1 \\ 1 & -2 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = -\frac{1}{9}\lambda^2 \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$

This is a matrix eigenvalue problem with the eigenvalue given by $\mu = -\lambda^2/9$. The solution for μ is arrived at by solving

$$\det \begin{pmatrix} -2-\mu & 1 \\ 1 & -2-\mu \end{pmatrix} = 0$$

with resulting quadratic equation

$$(2+\mu)^2 - 1 = 0$$

The solutions are $\mu = -1, -3$, and since $\lambda = 3\sqrt{-\mu}$, we have $\lambda = 3, 3\sqrt{3} = 5.1962$. These two eigenvalues serve as rough approximations to the first two eigenvalues π and 2π.

With A an n-by-n matrix, the MATLAB variable mu=eig(A) is a vector containing the n eigenvalues of the matrix A. The built-in function eig.m can therefore be used to find the eigenvalues. With n grid points, the smaller eigenvalues will converge more rapidly than the larger ones.

We can also consider boundary conditions on the derivative, or mixed boundary conditions. For example, consider the mixed boundary conditions given by $y(0) = 0$ and $y'(1) = 0$. The eigenvalues of Equation 7.4.1 can then be determined analytically to be $\lambda_i = (i - 1/2)\pi$, with i a natural number.

The difficulty we now face is how to implement a boundary condition on the derivative. Our computation of y'' uses a second-order method, and we would like the computation of the first derivative to also be second order. The condition $y'(1) = 0$ occurs on the right-most boundary, and we can make use of the second-order backward-difference approximation to the derivative that we have previously derived. This finite-difference approximation for $y'(1)$ can be written as

$$y'_{n+1} = \frac{3y_{n+1} - 4y_n + y_{n-1}}{2h}.$$

Now, the nth finite-difference equation was given by

$$y_{n-1} - 2y_n + y_{n+1} = -h^2 y_n,$$

and we now replace the value y_{n+1} using (7.11); that is,

$$y_{n+1} = \frac{1}{3}\left(2hy'_{n+1} + 4y_n - y_{n-1}\right).$$

Implementing the boundary condition $y'_{n+1} = 0$, we have

$$y_{n+1} = \frac{4}{3}y_n - \frac{1}{3}y_{n-1}$$

Therefore, the nth finite-difference equation becomes

$$\frac{2}{3}y_{n-1} - \frac{2}{3}y_n = -h^2\lambda^2 y_n$$

For example, when $n = 2$, the finite difference equations become

$$\begin{pmatrix} -2 & 1 \\ \frac{2}{3} & -\frac{2}{3} \end{pmatrix}\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = -\frac{1}{9}\lambda^2\begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$

The eigenvalues of the matrix are now the solution of

$$(\mu + 2)\left(\mu + \frac{2}{3}\right) - \frac{2}{3} = 0$$

Or

$$3\mu^2 + 8\mu + 2 = 0$$

Therefore, $\mu = (-4 \pm \sqrt{10})/3$, and we find $\lambda = 1.5853, 4.6354$, which are approximations to $\pi/2$ and $3\pi/2$.

Shooting method

We apply the shooting method to solve Equation 7.4.1 with boundary conditions $y(0) = y(1) = 0$. The initial value problem to solve is

$$\begin{aligned} y' &= z \\ z' &= -\lambda^2 y \end{aligned}$$

with known boundary condition $y(0) = 0$ and an unknown boundary condition on $y'(0)$. In fact, any nonzero boundary condition on $y'(0)$ can be chosen: the differential equation is linear and the boundary conditions are homogeneous, so that if $y(x)$ is an eigenfunction then so is $Ay(x)$. What we need to find here is the value of λ such that $y(1) = 0$. In other words, choosing $y'(0) = 1$, we solve

$$F(\lambda) = 0$$

where $F(\lambda) = y(1)$, obtained by solving the initial value problem. Again, an iteration for the roots of $F(\lambda)$ can be done using the Secant Method. For the eigenvalue problem, there are an infinite number of roots, and the choice of the two initial guesses for λ will then determine to which root the iteration will converge.

For this simple problem, it is possible to write explicitly the equation $F(\lambda) = 0$. The general solution to Equation 7.4.1 is given by

$$y(x) = A\cos(\lambda x) + B\sin(\lambda x).$$

The initial condition $y(0) = 0$ yields $A = 0$. The initial condition $y'(0) = 1$ yields

$$B = 1/\lambda$$

Therefore, the solution to the initial value problem is

$$y(x) = \frac{\sin(\lambda x)}{\lambda}$$

The function $F(\lambda) = y(1)$ is therefore given by

$$F(\lambda) = \frac{\sin \lambda}{\lambda}$$

and the roots occur when $\lambda = \pi, 2\pi, \ldots$.

If the boundary conditions were $y(0) = 0$ and $y'(1) = 0$, for example, then we would simply redefine $F(\lambda) = y'(1)$. We would then have

$$F(\lambda) = \frac{\cos \lambda}{\lambda}$$

and the roots occur when $\lambda = \pi/2, 3\pi/2, \ldots$

This page titled 7.4: Numerical Methods - Eigenvalue Problem is shared under a CC BY 3.0 license and was authored, remixed, and/or curated by Jeffrey R. Chasnov via source content that was edited to the style and standards of the LibreTexts platform.

Index

B

bisection method
2.1: Bisection Method
boundary value problems
7.1: Examples of Analytical Solutions

C

cubic spline interpolation
5.3: Cubic Spline Interpolation

D

double precision format
1.11: IEEE Double Precision Format

E

eigenvalue problems
7.1: Examples of Analytical Solutions
Euler method
7.2: Numerical Methods - Initial Value Problem

F

finite difference method
7.3: Numerical Methods - Boundary Value Problem

H

hexadecimal
1.5: 4-bit Unsigned Integers as Hex Numbers

I

initial value problem
7.1: Examples of Analytical Solutions
interpolation
5: Interpolation

L

Lagrange polynomials
5.1: Polynomial Interpolation
LU decomposition
3.2: LU Decomposition

M

machine epsilon
1.10: Machine Epsilon
midpoint rule
6.1: Elementary Formulas
multidimensional interpolation
5.4: Multidimensional Interpolation

N

Newton polynomial
5.1: Polynomial Interpolation
Newton’s method
3.5: System of Nonlinear Equations
Newton's method
2.2: Newton's Method
normal equations
4.2: Fitting to a Linear Combination of Functions

O

order of convergence
2.4: Order of Convergence

P

partial pivoting
3.3: Partial Pivoting
piecewise linear interpolation
5.2: Piecewise Linear Interpolation
pivot
3.3: Partial Pivoting

R

Richardson’s extrapolation
6.4: Adaptive Integration
roundoff errors
1.12: Roundoff Error Example

S

secant method
2.3: Secant Method
shooting method
7.3: Numerical Methods - Boundary Value Problem
Simpson’s rule
6.1: Elementary Formulas

T

trapezoidal rule
6.2: Composite Rules

V

Vandermonde matrix
5.1: Polynomial Interpolation
Vandermonde polynomial
5.1: Polynomial Interpolation

Detailed Licensing

Overview

Title: Numerical Methods (Chasnov)

Webpages: 53

All licenses found:

- CC BY 3.0: 66% (35 pages)
- CC BY 1.0: 24.5% (13 pages)
- Undeclared: 9.4% (5 pages)

By Page

- Numerical Methods (Chasnov) - *CC BY 1.0*
  - Front Matter - *CC BY 1.0*
    - TitlePage - *CC BY 1.0*
    - InfoPage - *CC BY 1.0*
    - Table of Contents - *Undeclared*
    - Licensing - *Undeclared*
    - Latex - *Undeclared*
  - 1: IEEE Arithmetic - *CC BY 1.0*
    - 1.1: Definitions - *CC BY 3.0*
    - 1.2: Numbers with a Decimal or Binary Point - *CC BY 3.0*
    - 1.3: Examples of Binary Numbers - *CC BY 3.0*
    - 1.4: Hex Numbers - *CC BY 3.0*
    - 1.5: 4-bit Unsigned Integers as Hex Numbers - *CC BY 3.0*
    - 1.6: IEEE Single Precision Format - *CC BY 3.0*
    - 1.7: Special Numbers - *CC BY 3.0*
    - 1.8: Examples of Computer Numbers - *CC BY 3.0*
    - 1.9: Inexact Numbers - *CC BY 3.0*
    - 1.10: Machine Epsilon - *CC BY 3.0*
    - 1.11: IEEE Double Precision Format - *CC BY 3.0*
    - 1.12: Roundoff Error Example - *CC BY 3.0*
  - 2: Root Finding - *Undeclared*
    - 2.1: Bisection Method - *CC BY 3.0*
    - 2.2: Newton's Method - *CC BY 3.0*
    - 2.3: Secant Method - *CC BY 3.0*
    - 2.4: Order of Convergence - *CC BY 3.0*
  - 3: System of Equations - *CC BY 1.0*
    - 3.1: Gaussian Elimination - *CC BY 3.0*
    - 3.2: LU Decomposition - *CC BY 3.0*
    - 3.3: Partial Pivoting - *CC BY 3.0*
    - 3.4: Operation Counts - *CC BY 3.0*
    - 3.5: System of Nonlinear Equations - *CC BY 3.0*
  - 4: Least-Squares Approximation - *CC BY 1.0*
    - 4.1: Fitting a Straight Line - *CC BY 3.0*
    - 4.2: Fitting to a Linear Combination of Functions - *CC BY 3.0*
  - 5: Interpolation - *CC BY 1.0*
    - 5.1: Polynomial Interpolation - *CC BY 3.0*
    - 5.2: Piecewise Linear Interpolation - *CC BY 3.0*
    - 5.3: Cubic Spline Interpolation - *CC BY 3.0*
    - 5.4: Multidimensional Interpolation - *CC BY 3.0*
  - 6: Integration - *CC BY 1.0*
    - 6.1: Elementary Formulas - *CC BY 3.0*
    - 6.2: Composite Rules - *CC BY 3.0*
    - 6.3: Local Versus Global Error - *CC BY 3.0*
    - 6.4: Adaptive Integration - *CC BY 3.0*
  - 7: Ordinary Differential Equations - *CC BY 1.0*
    - 7.1: Examples of Analytical Solutions - *CC BY 3.0*
    - 7.2: Numerical Methods - Initial Value Problem - *CC BY 3.0*
    - 7.3: Numerical Methods - Boundary Value Problem - *CC BY 3.0*
    - 7.4: Numerical Methods - Eigenvalue Problem - *CC BY 3.0*
  - Back Matter - *CC BY 1.0*
    - Index - *CC BY 1.0*
    - Glossary - *CC BY 1.0*
    - Detailed Licensing - *Undeclared*